Spring 5-1-2013

A Novel Methodology for Calculating Large Numbers of Symmetrical Matrices on a Graphics Processing Unit: Towards Efficient, Real-Time Hyperspectral Image Processing

Denise Renee Runnels
University of Southern Mississippi

The University of Southern Mississippi

A NOVEL METHODOLOGY FOR CALCULATING LARGE NUMBERS OF SYMMETRICAL MATRICES ON A GRAPHICS PROCESSING UNIT: TOWARDS EFFICIENT, REAL-TIME HYPERSPECTRAL IMAGE PROCESSING

by

Denise Renee Runnels

Abstract of a Dissertation
Submitted to the Graduate School
Of The University of Southern Mississippi
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

May 2013

ABSTRACT

A NOVEL METHODOLOGY FOR CALCULATING LARGE NUMBERS OF SYMMETRICAL MATRICES ON A GRAPHICS PROCESSING UNIT: TOWARDS EFFICIENT, REAL-TIME HYPERSPECTRAL IMAGE PROCESSING

by Denise Renee Runnels

May 2013

Hyperspectral imagery (HSI) is often processed to identify targets of interest. Many of the quantitative analysis techniques developed for this purpose mathematically manipulate the data to derive information about the target of interest based on local spectral covariance matrices. The calculation of a local spectral covariance matrix for every pixel in a given hyperspectral data scene is so computationally intensive that real-time processing with these algorithms is not feasible with today's general purpose processing solutions. Specialized solutions are cost prohibitive, inflexible, inaccessible, or not feasible for on-board applications.

Advances in graphics processing unit (GPU) capabilities and programmability offer an opportunity for general purpose computing with access to hundreds of processing cores in a system that is affordable and accessible. The GPU also offers flexibility, accessibility and feasibility that other specialized solutions do not offer. The architecture for the NVIDIA GPU used in this research is significantly different from the architecture of other parallel computing solutions. With such a substantial change in architecture it follows that the

paradigm for programming graphics hardware is significantly different from traditional serial and parallel software development paradigms.

In this research a methodology for mapping an HSI target detection algorithm to the NVIDIA GPU hardware and Compute Unified Device Architecture (CUDA) Application Programming Interface (API) is developed. The RX algorithm is chosen as a representative stochastic HSI algorithm that requires the calculation of a spectral covariance matrix. The developed methodology is designed to calculate a local covariance matrix for every pixel in the input HSI data scene.

A characterization of the limitations imposed by the chosen GPU is given and a path forward toward optimization of a GPU-based method for real-time HSI data processing is defined.

The University of Southern Mississippi

A NOVEL METHODOLOGY FOR CALCULATING LARGE NUMBERS OF SYMMETRICAL MATRICES ON A GRAPHICS PROCESSING UNIT: TOWARDS EFFICIENT, REAL-TIME HYPERSPECTRAL IMAGE PROCESSING

by

Denise Renee Runnels

A Dissertation
Submitted to the Graduate School
of The University of Southern Mississippi
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

Approved:

Dia Ali
Director

Chaoyang Zhang

Beddhu Murali

Bikramjit Banerjee

Ras Pandey

Adel Ali

Susan A. Siltanen
Dean of Graduate School

May 2013

ACKNOWLEDGMENTS

The writer would like to thank the dissertation director, Dr. Dia Ali, and the other committee members, Dr. Chaoyang Zhang, Dr. Beddhu Murali, Dr. Bikramjit Banerjee, Dr. Ras Pandey, and Dr. Adel Ali. I would especially like to thank Dr. Dia Ali for stepping in to the position of committee chair at such a late date and for his incredible support and encouragement throughout the process.

Additional thanks to Todd Hunter-Gilbert for his time spent working with me through some of the more frustrating code debugging sessions and to Dr. Andrew Strelzoff for helping get me started with this research. Special thanks also go to the Mobile Army Corps of Engineers for the sample data used in this research.

TABLE OF CONTENTS

ABSTRACT……ii

ACKNOWLEDGMENTS iii

LIST OF TABLES v

LIST OF ILLUSTRATIONS vi

LIST OF PSEUDOCODE viii

CHAPTER

I. INTRODUCTION 1

Motivating Problem
Contribution

II. BACKGROUND 8

Hyperspectral Imagery
GPU Computing History
Literature Review

III. METHODOLOGY 39

Hardware and Software Description
Approach

IV. RESULTS 89

Serial Results and Comparison Analysis
GPU Results and Analysis

V. CONCLUSIONS 114

GPU Advantages
GPU Limitations
Future Research
HSI Application
Summary

REFERENCES 123

LIST OF TABLES

Table

1. Timetable of GPU Development 27

2. Warp Scheduler Instruction Issue Timing for GT200 Architecture 45

3. Specification Comparison Between the G80, GT200, and Fermi Architectures 51

4. Quadro FX 4800 Technical Specifications 68

5. Block size limitations 71

6. Grid Layout for EW Sum Kernel 77

7. Memory Usage for EW Sum Kernel 79

8. Grid Layout for Calculating Covariance Matrix with Calculated EW Band Sum 81

9. Memory Usage for CovRX Kernel 84

10. Parameter Values for Empirical Study 87

11. System Specifications 88

12. GPU Component Processing Times for 512 X 512 X 36 Input and 17 X 17 EW 113

LIST OF ILLUSTRATIONS

Figure

1. Hyperspectral Image and Spectral Signature Graph 11
2. CPU / GPU Design Comparison 42
3. NVIDIA Streaming Processor 43
4. GT200 Streaming Multiprocessor 44
5. Thread Processing Cluster in the GT200 46
6. GT200 GPU Memory Architecture 47
7. Relative Size and Access Speeds for Different GT200 Memory Types ... 49
8. Top Level View of GT200 Architecture 50
9. CUDA Compute Capability 1.3 Logical Grid and Block Configuration Options 54
10. Relationship of Thread and Memory Hierarchies. 56
11. CUDA and GPU Hardware Relationship Diagram 57
12. RX Dependency and Interaction Graph 64
13. Illustration of BIP, BIL, and BSQ Data Formats 73
14. Total Serial Processing Times for the RX Algorithm 90
15. Total Serial Processing Time Comparison as EW Size Changes 91
16. Total GPU Processing Times for the RX Algorithm. 92
17. Total GPU Processing Time Comparison Based on EW Sizes 92
18. Serial / GPU Processing Time Ratio for Varying Numbers of Bands 94
19. Serial / GPU Processing Time Ratio for Varying EW Sizes 94
20. Total GPU Processing Time 95
21. GPU Processing Time for Different Kernel Block Layouts 97
22. CUDA Initialization Times 98

23. CUDA Initialization Time for Different Sizes of Input Data. 98

24. CUDA Initialization Times as EW Sum Block Numbers of Threads Vary 99

25. CUDA Initialization Times as CovRX Block Numbers of Threads Vary.. 100

26. EW Sum Kernel Times..... 101

27. EW Sum Kernel Processing Time with Different Numbers of Threads .. 102

28. EW Sum Kernel Processing Times with Different Block Width : Height Ratios..... 103

29. EW Sum Kernel Times for Different Size EWs and Numbers of Bands. 104

30. CovRX Kernel Times for Different Numbers of Bands 104

31. CovRX Kernel Times with Different Numbers of Threads and EW Sizes..... 105

32. CovRX Kernel Times with Different EW Sizes and Numbers of Bands.. 106

33. Data Transfer Times From Host to Device..... 107

34. Result Image Data Transfer Times From Device to Host..... 108

35. Quotient of the Data-to-Device Time with the Results-to-Host Time..... 109

36. Data Transfer Times and Sizes 110

LIST OF PSEUDOCODE

Pseudocode

1. Serial Program Design Using Cholesky Decomposition. 62
2. Algorithm for GPU Parallelization of EW Sum. 78
3. Algorithm for GPU Parallelization of the Covariance Matrix Calculation. . 82
4. Final Calculation of the RX Algorithm. 83

CHAPTER I

INTRODUCTION

"Remote sensing is the science and art of obtaining information about an object, area, or phenomenon through the analysis of data acquired by a device that is not in contact with the object, area, or phenomenon under investigation" (Lillesand, Kiefer, and Chipman 2004, 1). Remote sensing technologies provide the world's scientists, military, farmers, and many other end users with useful information for studying the mysteries of the oceans, finding the location of military threats, determining which fields need fertilizing; as many varied types of information as the different types of users. In this age of technology, remote sensing solutions have become increasingly sophisticated and are deployed on platforms as varied as the data collected; from unmanned underwater vehicles to satellites and many wide-ranging platforms in between. The information garnered from these remote sensing systems begins as data that must go through various processing steps to become useful and actionable information.

This data is often processed on-board the platform and only the processed results are stored or transmitted to the user. However, in some cases the sensor data collected requires off-loading to a more powerful computing system for processing. Such is the case with hyperspectral imagery (HSI), a type of imagery data collected with an imaging spectrometer. Imaging spectrometers are "designed to collect data on the spectral as well as the spatial characteristics of the imaged scene" (Schott 2007, 212).

Hyperspectral image data is similar to true color digital imagery, but instead of being comprised of data from three visible broadband wavelengths (blue, green, and red) it is comprised of many narrowband wavelengths (over 200 in some cases) within a range from the blue through the infrared (90.4-2500nm) portions of the electromagnetic spectrum (Jia, Qian, and Ji 2008). Whereas computing capabilities and image processing algorithms for true color digital imagery are currently mature and efficient for most uses, hyperspectral image processing requires significantly more computing power than typical digital imagery. Nevertheless, the additional information contained within HSI data justifies the use of additional resources.

This is not a new technology. Indeed, the basis for the current digital technology acquiring remotely sensed radiometric data dates back to the late 1970s with the Advanced Very High Resolution Radiometer (AVHRR) (Hastings and Emery 1992) currently deployed on the Polar Orbiting Environmental Satellite (POES), administered by the National Oceanic and Atmospheric Administration (Office of Satellite Operations). The many advances made with radiometric imagers since the late 1970s have led to sophisticated hyperspectral imaging sensors that collect and store three dimensional volumetric data sets of several hundreds of megabytes, or even gigabytes, per data collection (Robila and Busardo 2011). Although the imaging technology is not new, the ability to manage gigabytes of data on-board a sensing platform has only become an option with recent developments in memory storage and computing solutions that weigh less, require less physical space and require less power.

Motivating Problem

As with so many technological advancements, no sooner is one challenge successfully overcome than the end users push for evermore speed, evermore space, evermore computing power. So it is with HSI remote sensing technologies. Recent advances in memory capabilities have led to yet more sophisticated HSI sensors that collect more data at higher spatial and spectral resolutions. These increasingly large, volumetric data sets stress even the current high capacity systems.

Although the central processing unit (CPU) computational capabilities of these remote sensing systems have increased along with the memory capabilities, the nature of the processing requirements for HSI limit the feasibility of on-board processing: to provide real-time useful information for the end user while also reducing on-board storage needs.

Hyperspectral imagery is often processed to identify targets of interest. For a farmer, that target might be a field of crops that need fertilizing; for the military, the target might be a particular type of vehicle. The quantitative analysis techniques developed for this purpose use computationally intensive computer-based algorithms to process the volumetric data. Specifically, many of the algorithms that mathematically manipulate the data to derive information about a target of interest require the calculation of a covariance matrix. More importantly, the algorithms with the best probability of detection, and simultaneous low false alarm rate, require a local covariance matrix for each pixel in the data scene. Given a 512×512 scene with 32 spectral channels, 262,144 matrices each with

1024 elements are computed. These covariance matrix calculations are by far the most computationally intensive component of the algorithms in this class of HSI algorithms (Kwatra and Han 2010). Similarly, the de facto standard anomaly detection algorithm RX (Reed-Xaoli) shares this computationally expensive requirement (Matteoli, Diani, and Corsini 2010, 1).

Currently, many HSI programs must limit the time in the air collecting data or subsample the imagery due to lack of sufficient storage. An alternative solution is to off-load the data via downlink. However, this solution also requires subsampling due to downlink bandwidth limitations. While the subsampled solution is sufficient for many applications, in other cases it is preferable to process the full spatial and spectral data set without loss of data. Another solution is to process the data on-board, yet as mentioned earlier this requires significant computational capabilities. Specialized hardware solutions that utilize a field programmable gate array (FPGA), such as the Airborne Real-time Cueing Hyperspectral Enhanced Reconnaissance (ARCHER) program implemented with the Compact Airborne Spectral Exploitation (CASE) processor, are cost prohibitive and not flexible for multiple diverse applications (Novasol 2007a; Novasol 2007b).

Nevertheless, processing data as fast as possible is ever a goal of technological progress, beyond the benefit of requirements for less storage, faster processing provides more processed data in less time, which leads to more information acquired which in turn allows for more informed decisions. High performance computers (HPC) such as supercomputers or computer clusters

have the computational power to process HSI data faster than the average desktop with the flexibility not afforded by an FPGA solution; however, these systems are not readily available to the average remote sensing laboratory. Likewise, traditional HPC resources are too large, too heavy, and require too much power to be installed on a typical HSI remote sensing platform or in an average remote sensing analyst's laboratory. Thus a solution for fast processing on a desktop system is beneficial to those researchers without access to traditional HPC resources. Similarly, many airborne remote sensing platforms can accommodate a desktop-sized computer should it prove capable of on-board processing.

On-board processing would: reduce the on-board storage requirements, thereby reducing the weight and power requirements of the system or allowing systems to collect longer; reduce the size of the downlinked data product; and provide real-time (or near real-time) information that would allow immediate response such as is needed for security and defense applications.

Contribution

Improvements in disk memory and CPU processing capabilities are not the only relevant improvements in the recent past. The gaming industry's successful advances with graphics processing units (GPU) moved the GPU from special purpose rendering processors for gaming to a more general purpose parallel processing platform (Kanter 2008) that can be installed in many modern desktop computers. ATI and NVIDIA are currently the two preeminent GPU vendors. NVIDIA however has developed the "most comprehensive and

consistent approach to general purpose computation" (Kanter 2008, 1) particularly with the introduction of the Compute Unified Device Architecture (CUDA™) Application Programming Interface (API).

CUDA is a parallel computing architecture and programming model that allows a software programmer to capitalize on the computational power of the GPU (Kirk 2007). This software architecture is designed to align very closely with the NVIDIA GPU hardware architecture so that an optimally written CUDA program will optimally utilize the GPU processing capabilities.

The architecture of the NVIDIA GPU is significantly different from the architecture of other parallel computing solutions. Specifically, the memory architecture differs from either traditional shared memory supercomputers or distributed memory supercomputers and clusters. GPU hardware was designed for the special purpose of rendering graphics. Thus the specialized solution will implement primitives such as triangles and vectors optimized for matrix math operations. Similarly, the texture memory hierarchy is geared toward optimizing access for 2D spatial locality with four-part texture elements: texels (Farber 2009). Therefore, although the GPU is designed to manage thousands of compute threads simultaneously, not all processing problems will efficiently map to this solution.

With such a significant change in architecture it follows that the paradigm for programming graphics hardware is significantly different from the traditional serial and parallel software development paradigms. Leveraging the natural correlation between imagery and computer graphics, it is reasonable to expect

speed improvements typical of a traditional parallel solution for hyperspectral image processing using a GPU. Thus, the contribution for this research is the development of a methodology for reshaping the solution space of traditional HSI algorithm implementations to apply to the GPU programming framework. The GPU framework is described and a method for dividing the HSI processing algorithm to map to this framework is given. Although the developed methodology is a naïve approach, the research contributes a characterization of the limitations imposed by the NVIDIA GPU with compute capability 1.3 and earlier. While these limitations are significant, optimization opportunities for dividing the problem are possible and suggestions are given for future work. Similarly, newer technology with NVIDIA compute capability 2.0 is available that mitigates several of the limitations imposed by compute capability 1.3. This research therefore defines a path forward toward optimization of a GPU-based method for real-time HSI data processing using both new techniques for dividing the problem and leveraging the capabilities offered by the newer hardware capabilities.

A representative solution for calculating on a GPU the thousands of covariance matrices required by the RX algorithm is given. The speed performance of the unoptimized GPU-based implementation improves by two orders of magnitude when compared with an unoptimized serial implementation of the algorithm.

CHAPTER II

BACKGROUND

Hyperspectral Imagery

Remote sensing technologies have been in use for many decades. One of the first documented examples of remote sensing dates back to 1904 when small cameras were attached to homing pigeons that would fly over an area of interest while the camera periodically captured an image (Jensen 2007). From black and white film cameras strapped to a pigeon the technology has progressed to digital imaging of a few broad spectral bands covering several kilometers in a single swath to hundreds of narrow spectral bands covering several centimeters in a single swath from a satellite as the system platform.

The first systematic satellite images of earth came from NASA's Television Infrared Observation Satellite (TIROS-1) (Hastings and Emery 1992). A series of sensors deployed on NASA and NOAA satellites moved from the analog television camera to scanning radiometers, very high resolution radiometers, and the digital advanced VHRR with five thermal channels before reaching the first hyperspectral sensor deployed on a satellite (Hastings and Emery 1992). Currently many hyperspectral imaging (HSI) sensor programs are in operation and provide HSI data to a variety of researchers. Airborne Visible Infrared Imaging Spectrometer (AVIRIS), Hyperion, Hyperspectral Digital Imagery Collection Experiment (HYDICE), Compact Airborne Spectrographic Imager (CASI), and Moderate Resolution Imaging Spectrometer (MODIS) are a few

spectral sensors and spectral sensor programs providing HSI data for research purposes (Borstad; Huertas and Nevatia; NASA 2013a; NASA 2013b; USGS).

Hyperspectral Imagery is defined as imagery collected with narrow and contiguous spectral bandwidths (Shippert 2003). The high spectral resolution of a narrow bandwidth in conjunction with contiguous measurements is what differentiates HSI from multispectral imagery (MSI). While some MSI sensors may collect in relatively narrow bandwidths, they are designed to collect at discrete locations of the electromagnetic spectrum with gaps between each channel (band) collected. Additionally, many definitions of HSI differentiate HSI and MSI by the numbers of bands collected, where HSI data consists of significantly more bands of data than MSI data. This high spectral resolution combined with the high spatial resolution of today's technologies result in very large volumetric data sets leading to analysis difficulties.

Hyperspectral image analysis generally comprises three processing phases; preprocessing, image enhancement, and information extraction (Rees 2003). The preprocessing phase involves correction for errors in the data due to the specific sensor system that collected the data. These are primarily radiometric and geometric errors and are addressed using calibration and registration techniques, respectively (Rees 2003). Radiometric error correction is also used to remove atmospheric propagation effects, especially for imagery collected from a satellite platform.

The image enhancement phase includes methods to improve the usability of the data. These techniques often include transforming the data to a common

domain, i.e. reflectance, radiance, or digital count. Image enhancement may also include removing unusable data bands such as water absorption bands in the vicinity of ± 940nm wavelength on the electromagnetic spectrum (Schott 2007).

While the first two phases of the HSI analysis process are important, the information extraction phase is addressed in this project. Several algorithms have been developed to identify patterns in HSI data, many of which are quantifiable.

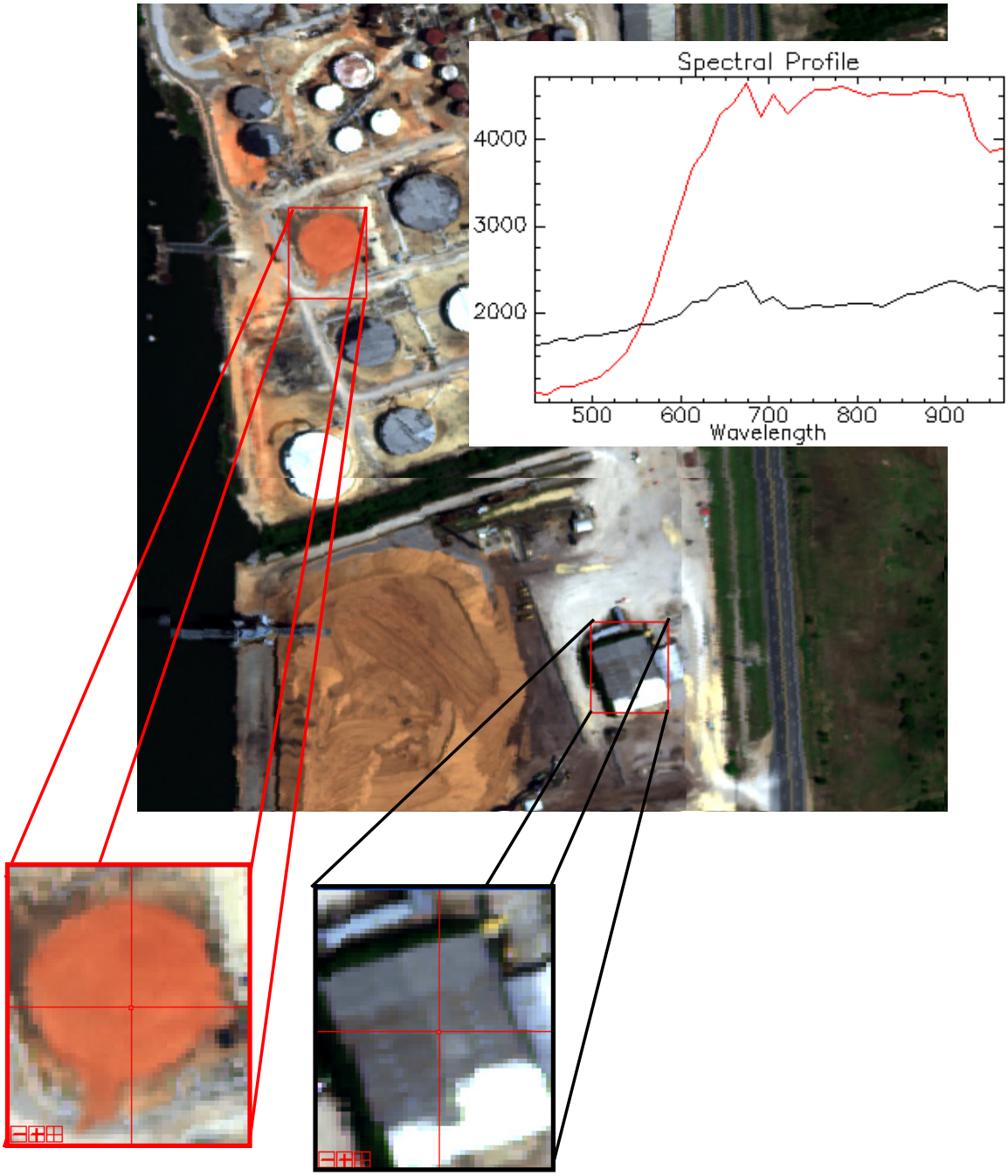

Figure 1. Hyperspectral Image and Spectral Signature Graph. Data provided by Mobile Army Corps of Engineers.

Hyperspectral Image Processing Algorithms

Algorithms to exploit HSI data for information extraction have advanced much as the sensor systems have advanced. Many of these algorithms are

based on radar and other signal processing techniques applied to the spectral domain rather than the spatial or temporal domains. It is the curve provided by the contiguous nature of HSI spectral bands that is exploited in HSI analysis. Figure 1 is a true color display of a hyperspectral image (top) with spectral signatures plotted (top right) for two different materials, one from a silo and one from the roof of a building. The 36 band data provided by the Mobile Army Corps of Engineers was collected with the CASI sensor. Intensity in digital numbers is plotted on the Y axis and wavelength in nanometers is plotted on the X axis.

An HSI sensor captures photons reflected by material in the scene. This spectral reflectance signature (curve) exhibits specific characteristics that allow the material to be identified, provided the reflectance signature is in the signature library used for analysis. This technique is used for target detection and identification. Similarly, the spectral signatures of picture elements, or pixels, in a scene are often compared to the spectral signatures of the surrounding pixels for spectral anomaly detection, in which case a spectral library is not needed. Alternatively, many HSI analysis algorithms simply perform spectral classification in which pixels are clustered together with other “like” pixels in the scene. Thus, two broad areas for HSI algorithm development include classification algorithms and detection algorithms.

Classification Algorithms

“Image classification is the process of making quantitative decisions from image data, grouping pixels or regions of the image into classes intended to represent different physical objects or types” (Rees 2003, 296). Hyperspectral

image classification algorithms are used for applications such as environmental mapping, land use change detection, and abundance estimation (Harsanyi and Chang 1994). Both supervised and unsupervised classification techniques are used. Supervised classification requires a *man-in-the-loop* to provide training data for the algorithm. That is, the analyst specifies pixels representative of the classes of interest as input to the algorithm. The algorithm then classifies all pixels in the area of interest based on the given pixel classes using a discriminate function such as the Euclidean distance. An unsupervised classification algorithm does not need training data as input.

K-Means

The K-means or isodata algorithm is the most commonly used example of an unsupervised classification algorithm (Rees 2003). This algorithm is iterative, seeded in the first step by choosing cluster “centers” by calculating initial class means based on arbitrarily assigned pixels (Filho et al. 2003; RSI 2003). All pixels in the area of interest are then assigned a class using a minimum distance technique, often the Euclidean distance (Filho et al. 2003). Each subsequent iteration recalculates the cluster centers based on the pixels currently assigned to the class, then reclassifies the pixels accordingly (RSI 2003). The number of clusters is fixed, or alternatively may be defined by the analyst.

Detection Algorithms

Over the past few decades many HSI detection algorithms have been developed. The majority of these techniques can be divided into two general types of algorithms; those that use a geometric approach and those that use a

statistical approach (West et al. 2005). The most common HSI detection algorithms apply the statistical, or stochastic, approach and require calculating a covariance matrix of the background clutter, a motivating factor for this research (Manolakis et al. 2009). Thus, the detection algorithm discussion here focuses on algorithms that use the spectral covariance matrix. The single exception, spectral angle mapper, is a geometric approach exemplifying an HSI detection algorithm that is not significantly limited by today's typically available computational resources.

Target Detection

Target detection algorithms look to determine if a specified target of interest is present in the HSI data scene. These algorithms require a reference spectral signature of the target of interest with which to compare each pixel in the scene. Depending on the target, the reference spectrum of a known material may be available through model simulation or from a reference library usually developed with laboratory spectrometers. In this case, it is very important that the preprocessing and image enhancement phases in the HSI processing chain are performed well, especially the image enhancement technique to put the image and the reference signature in the same domain for comparison. Alternatively, the reference signature may come from the data being processed via an analyst, in which case the signature is already in the domain of the data. Data-derived signatures may or may not be of a known object or material. These methods not only provide detection information, but if used with known reference signatures may also provide identification of the detected material (Kruse 1994). The

algorithm descriptions here assume that a reference signature is available and that the data and reference are in a common domain.

Spectral Angle Mapper

Many geometric, or deterministic, algorithms employ a subspace model to characterize the background (Manolakis et al. 2009). However, the "spectral angle mapper (SAM) is one of the most widely used algorithms for making simple comparisons between spectral vectors" (Schott 2007, 433). The SAM algorithm compares two spectra by calculating the spectral angle between them (Kruse et al. 1993). This simple deterministic algorithm takes the arccosine of the dot product of the reference and pixel of interest spectra and can be expressed as (Girouard et al. 2004):

$$\cos(\alpha) = \left(\frac{\sum_{i=1}^{n} \mathbf{p}_i \mathbf{r}_i}{\left(\sqrt{\sum_{i=1}^{n} \mathbf{p}_i^2}\right)\left(\sqrt{\sum_{i=1}^{n} \mathbf{r}_i^2}\right)} \right) \qquad \begin{cases} no\ target, \cos(\alpha) < t \\ target, \quad \cos(\alpha) \geq t \end{cases} \qquad \text{Eq. 1}$$

where $\mathbf{p}$ is the pixel spectrum, $\mathbf{r}$ is the reference spectrum, and n is the number of bands. A threshold value t provides a binary indicator for the presence or absence of the target of interest based on the reference spectrum.

In addition to using SAM with a reference signature for target detection, SAM can also be used for anomaly detection and classification without a known reference spectrum (Schott 2007). The relative simplicity of this algorithm combined with its flexible usages make it very popular, however the detection and classification performance is poor compared to the more complex algorithms discussed below (Yuan and Niu 2007).

Spectral Matched Filter

"The most widely used algorithm for hyperspectral target detection is the matched filter (Manolakis et al. 2007, 529). This stochastic approach was originally designed for radar signal processing to maximize the signal-to-noise ratio (SNR) when the most significant source of variability is noise, in which case the covariance calculated is a noise covariance (Schott 2007). In adapting this algorithm for HSI applications it is not the noise that dominates the variability in the scene, rather it is variation of the scene itself so that it is the signal-to-clutter ratio (SCR) that is maximized (Schott 2007; Shippert 2003). The covariance calculated in this case is the spectral covariance of the background, which includes both scene and noise variability and is applied to suppress the response from any pixel spectra that do not match the reference spectrum (Nasrabadi 2007, 72). The spectral matched filter is written as:

$$SMF(\mathbf{p}) = (\mathbf{r}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{p}-\boldsymbol{\mu}) \qquad \begin{cases} no\ target, SMF(\mathbf{p}) < t \\ target, \quad SMF(\mathbf{p}) \geq t \end{cases} \quad \text{Eq. 2}$$

where $\mathbf{p}$ is the pixel spectrum, $\mathbf{r}$ is the reference spectrum, and $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ are the background mean and covariance matrix, respectively.

Generalized Likelihood Ratio Test

Kelly (1986) originally introduced the generalized likelihood ratio test in response to the challenge of finding a radar signal present in the radar data being processed while maintaining a constant false alarm rate (Kelly 1986). "The GLRT is [now] known to be the benchmark detector for a multivariate complex-Gaussian noise environment" (Pulsone and Zatman 1999). While the GLRT is more computationally complex than some other HSI detection algorithms, the

detection performance is often better particularly in cases in which the distribution is Gaussian, and therefore a popular choice despite the extended processing time (Theiler, Foy, and Fraser 2005). The GLRT is expressed as (Schott 2007):

$$GLRT(\mathbf{p}) = \frac{\left[(\mathbf{r}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{p}-\boldsymbol{\mu})\right]^2}{\left[(\mathbf{r}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{r}-\boldsymbol{\mu})\right]\left[1+\frac{1}{N}(\mathbf{p}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{p}-\boldsymbol{\mu})\right]} \begin{cases} no\ target, GLRT(\mathbf{p}) < t \\ target, \quad GLRT(\mathbf{p}) \geq t \end{cases} \quad \text{Eq. 3}$$

where, as above, $\mathbf{p}$ is the pixel spectrum, $\mathbf{r}$ is the reference spectrum, $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ are the background mean and covariance matrix, respectively, and N is the number of pixels in the estimation window used to calculate the covariance matrix.

Anomaly Detection

RX

The RX anomaly detector, named after developers Reed and Xaoli, is based on the GLRT (Nasrabadi 2009, 159) and is the "benchmark anomaly detection algorithm" for hyperspectral imagery (Matteoli, Diani, and Corsini 2010, 1). The RX algorithm has been modified in many ways with the intent of optimizing the calculations or detection performance. However, RX in the *simplest form* is written as (Alonso and Malpica 2009):

$$RX(\mathbf{p}) = (\mathbf{p}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{p}-\boldsymbol{\mu}) \qquad \begin{cases} no\ target, RX(\mathbf{p}) < t \\ target, \quad RX(\mathbf{p}) \geq t \end{cases} \quad \text{Eq. 4}$$

where $\mathbf{p}$ is the pixel spectrum, and $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ are the background mean and covariance matrix, respectively.

A Common Form

It is noticeable when looking at the RX algorithm that it is effectively the Mahalanobis Distance squared and that this form is common among the three stochastic examples given (Theiler, Foy, and Fraser 2005). Similarly of note, the most computationally demanding calculation is that for the covariance matrix. The background spectral covariance matrix entry calculation for bands b_i and b_j at pixel of interest p is given as:

$$\operatorname{cov}\left(b_i, b_j\right)_p = \left(1/N\right)\sum_{k=1}^{N}\left(\mathbf{b}_{i_k} - \boldsymbol{\mu}_i\right)\left(\mathbf{b}_{j_k} - \boldsymbol{\mu}_j\right) \qquad \text{Eq. 5}$$

where $\boldsymbol{\mu}$ is the background mean, and $\mathbf{b}_i$ and $\mathbf{b}_j$ are bands i and j, respectively and N is the number of pixels in the estimation window used to compute the means.

The covariance matrix may be calculated as a global estimation which characterizes the background of the entire scene in a single matrix. A single covariance matrix calculation is negligible in terms of computation costs and is readily accomplished with typical desktop computing power. Using a global covariance with anything other than a relatively homogeneous scene, however, results in poor detection performance (Banerjee, Burlina, and Diehl 2009, 190). Alternatively, calculating the covariance to characterize the local background in the relatively immediate vicinity of the pixel of interest offers improved detection results (Manolakis et al. 2007) at the expense of significant computation cost as the covariance must be calculated for each pixel of interest rather than once for the entire scene.

Algorithms and Limited Resources

Most HSI algorithms developed for target detection are based on several assumptions that are often invalid in practical applications (Manolakis et al. 2007). The resulting detection performance is consequently impacted. Unfortunately, algorithm developers must work to optimize detection performance while under the constraint of limited computational resources.

One approach to address limited resources reduces the dimensionality of the data as part of the pre-processing and image enhancement stages, beyond removing unusable noise bands such as the water absorption bands. The popular principal component analysis (PCA) method is designed to transform HSI data such that uncorrelated bands are identified. Additional analysis is then performed with this reduced data set of chosen PCA bands. While this addresses limited computational resources it comes with a detection performance tradeoff, much like the global covariance calculation (Matteoli, Diani, and Corsini 2010; Joevivek, Hemalatha, and Soman 2009).

Despite decades of HSI algorithm development, a *silver bullet* solution has not yet been developed that performs equally well in every scenario, or in fact even in most scenarios (Alonso and Malpica 2009). An implementation of the covariance based algorithms that does not require reduced dimensionality could prevent underutilization of the captured data for improved detection performance (Kruse 1994). A GPU solution may offer this implementation without the computational delays of current systems.

GPU Computing History

According to Plato, necessity is the mother of invention, and according to many academics working to solve computationally intensive problems it is necessary to perform many gigaflops at low cost, i.e. without large CPU clusters or super computers. Thus when non-programmable graphics accelerators became programmable, a few avant-garde scientists thought to exploit the raw compute power afforded by the specialized graphics hardware for non-graphical purposes (Brookwood 2009). With the introduction of NVIDIA's GeForce 3 in 2001 these enthusiastic high performance computing (HPC) scientists were provided a means to perform programmable pixel shading, albeit using graphics-specific APIs such as OpenGL and DirectX (Glaskowsky 2009; Macedonia 2003).

The largest group of consumers for computer graphics hardware is in the video game industry (LoPiccolo 2003). Thus, the majority of the effort put forth to improve graphics solutions is geared toward improving graphics performance in video games. Considering the computer graphics market exceeded $67 billion in 2011, computer graphics development is well funded (Peddie 2011).

In the early 2000's it became obvious to "graphics industry observers" that a trend was developing toward a plateau in "pixel / vertex / triangle growth" (Dipert 2005, par. 1). This plateau was due in part to a corresponding plateau in CPU clock speeds. Sutter points out that in early 2003 the trend of ever increasing CPU clock speeds experienced a "sharp flattening" (Sutter 2005, 17). This, combined with ever improving graphics capabilities, allowed the GPU to

process graphics-related data faster than the CPU could supply it (Thompson, Hahn, and Oskin 2002). While improving functions such as antialiasing would employ some of the additional compute capability, there was concern that the GPU would be underutilized if the current trends of the time continued because “games [were] CPU-limited" (Dipert 2005). Thus graphics architecture designs began to emphasize flexibility, moving away from the earlier fixed function pipelines and into the realm of programmability (Thompson, Hahn, and Oskin 2002).

How convenient then that a select few HPC scientists proved the efficacy of applying the massively parallel capabilities provided by the GPU to some HPC problems (Tulloch 2006). Even though the possibility of using GPUs to address HPC problems was demonstrated on the early GPUs, it was nevertheless extremely challenging to implement (Macedonia 2003). The problem needed to be mapped to primitives such as triangles and pixels supported by the graphics APIs and adapted to utilize the fixed function pixel shading pipeline (Buck 2004; Thompson, Hahn, and Oskin 2002). This meant that the programmer not only had to have extensive knowledge of the computational science problem being explored, but also a strong understanding of graphics primitives and pipeline operations (Harris 2005). Thus, advancements in software APIs were also needed to make applying HPC problems to GPU solutions feasible. In fact, development in three primary areas was needed: the interconnect between the CPU and GPU; the software APIs used to program the GPU; and the GPU hardware.

Central Processing Unit / Graphics Processing Unit Interconnect

Because the GPU is a co-processor with the CPU, it would matter very little how fast the GPU could process volumes of data if the data bus between the CPU and GPU could not provide effective data transfer speeds (Thompson, Hahn, and Oskin 2002). In the early days the PCI bus offered only a unidirectional peak 133MB/s bandwidth, which resulted in such latencies that processing may as well have been performed on the CPU (Dipert 2005). However, the Intel AGP8x (accelerated graphics port) with 2.1GB/s bandwidth in the GPU to CPU direction provided adequate speeds for feasibility studies (Macedonia 2003). In practice this soon became the bottleneck for general purpose GPU applications. The first version of PCI Express (PCIe 1.0 x4) made significant improvements offering four dual wire bus lanes, each lane simultaneously providing 256MB/s of bidirectional bandwidth resulting in a total 1024MB/s two-way peak bandwidth (Dipert 2005). Since the 2003 release of PCIe 1.0 the PCI Express interconnect speeds continued to increase, first by adding lanes with x8 and x16 solutions. Then in 2007 the increase doubled with PCIe 2.0 when the clock speed was increased from 250MHz to 500MHz (Sherwin 2007). Interestingly, work to double the PCIe 2.0 speeds with the PCIe 3.0 specification began as soon as 2.0 was completed, but it was not until late 2011 to mid 2012 that AMD and NVIDIA, respectively, released the first PCIe 3.0 compatible GPUs (Del Rizzo 2012; Erskine and Kanas 2011).

Graphics Processing Unit Application Programming Interface

The general purpose GPU pioneers provided proof of concept that the parallel architecture of the GPU could be leveraged for high performance computing problems. However, they also demonstrated that the feasibility for doing so rested in development of a less graphics-specific programming interface. While Microsoft's DirectX 9.0 specification provided a stepping stone with 32-bit floating point capability, it was nevertheless a graphics-specific API (Macedonia 2003).

The leaders in the graphics hardware industry, namely ATI (now AMD) and NVIDIA, recognized the new market and began working to address the issues facing general purpose GPU developers (Tulloch 2006). This new direction became known as General Purpose computation on Graphics Processing Units (GPGPU) (GPGPU). Early attempts at developing a higher level interface include Close To Metal (CTM) created by AMD, Cg developed by NVIDIA, and Brook developed at Stanford University (AMD 2006; Fernando and Kilgard 2003; Buck et al. 2004).

CTM, AMD's initial attempt to abstract the graphics-centric aspects of programming the GPU, was a low level language that provided programming flexibility beyond the fixed function graphics pipeline yet it did little to alleviate programming difficulty for the non-graphics expert (Kingyens and Steffan 2011). The Cg API developed by NVIDIA in collaboration with Microsoft, was a higher level language based on the C programming language and also offered a step forward for GPU programmability. Cg provided developers with the tools needed

“to implement any GPU program rather than restricting developers to a predefined framework designed specifically for shading computations” (Fernando and Kilgard 2003, xxxi). Nevertheless, Cg still required computations to be mapped to shading operations on graphics primitives (Buck et al. 2004).

Researchers at Stanford University were some of the first to exploit GPUs for general purpose computing and found that working with the lower level languages and APIs available at the time were unwieldy and limiting. With the development of Brook, Stanford attempted to bridge the gap between graphics experts and HPC parallel programming specialists by introducing a streaming model to GPU programming (Buck et al. 2004). This model uses terminology and concepts such as streams and kernels, which are familiar to HPC programmers, and applies them to GPU terms and concepts such as texture and pixel shader (Abi-Chahla 2008; Hager and Wellein 2011, 5). Brook specifically abstracted memory management, data-parallel operations, and reductions (Buck et al. 2004). These are all important aspects of high performance computing.

The binary generated by the Brook compiler was designed to link via run-time libraries to available graphics APIs such as DirectX and OpenGL. While Brook provided programming accessibility to the power of the GPU, this link to a 3D API could add a significant workload to the program. The most significant issue facing Brook, however, was incompatibility, a result of GPU vendor’s frequent driver updates (Abi-Chahla 2008).

Interestingly, two developers on the Brook development team accepted positions with AMD and NVIDIA to develop the next stepping stones to feasible

GPGPU computing. Thus the AMD Stream SDK, originally called Brook+ and CUDA developed by NVIDIA both have roots in Brook (Kowaliski 2008). Like Brook, these next generation APIs extend the C programming language and work on the stream programming model (Abi-Chahla 2008; Kowaliski 2008).

Patti Harrell, AMD Stream Computing Director, points out in a phone interview with Cyril Kowaliski that AMD brought the Brook project “in-house, clean[ed] it up” and replaced the link to other 3D APIs with a “new back-end that talks to [AMD’s] lower-level interface” (Kowaliski 2008, par. 8). This addressed compatibility issues with AMD GPUs as well as the excess overhead of the larger 3D APIs. Ian Buck, the lead Brook designer at Stanford, joined NVIDIA to do much the same thing developing the CUDA API for compatibility with NVIDIA’s programmable GPUs (Dang 2009).

At the same time that AMD and NVIDIA were developing the Stream SDK and CUDA, Apple announced the OpenCL specification to the Khronos Group. Khronos formed a Compute Working Group, of which AMD and NVIDIA are part, to create the OpenCL standard for parallel computing, not only for GPUs but across multiple GPUs and CPUs (Martellaro 2008). Adoption for the OpenCL standard is happening relatively slowly due to concerns that the portability capabilities negatively impact performance. Many studies have compared OpenCL implemented programs with CUDA programs and only recently have those comparisons come back with positive results for OpenCL performance (Fang, Varbanescu, and Sips 2011).

Graphics Processing Unit

While the opportunity for utilizing the significant calculation power of the GPU for something other than graphics is promising, the HPC community is a relatively small part of the GPU market. As such GPU makers work to continue improving graphics performance as well as GPU programmability (NVIDIA 2009a). Beyond programmability, HPC computing required functionality not necessary for graphics uses. Specifically, early GPUs did not offer floating point operations, much less the double precision necessary in most HPC applications (Dang 2009).

The earliest programmable GPU was NVIDIA's GeForce 3 released in 2001. This GPU had very limited programmability and did not include support for floating point operations (Glaskowsky 2009). In late 2002 using DirectX 9.0, ATI (now AMD) provided the first GPU with floating point programmability, although floating point support was limited (Abazovic 2002).

While programmability with support for floating point operations were essential for utilizing GPUs in HPC applications, it was not until the GPU hardware was modified such that memory access accommodated a unified computational architecture that the current advances to GPU computing were possible (Kanter 2008). This unified architecture provided the framework necessary to support the C programming language, in particular memory pointers (Dang 2009; Glaskowsky 2009). In 2006 GPU hardware offered a unified computational architecture, stream computing, full floating point capability, and shared memory (Chu 2010; Brookwood 2009; Glaskowsky 2009; Dang 2009).

The shared memory allowed inter-thread communication, though only a single multi-threaded kernel could execute at any given time (NVIDIA 2009a).

Table 1

Timetable of GPU Development

	pre 2001	2001	2002-2003	2006	2007	2008	2009 - 2010
Bus Width	< 256-bit	256-bit		384-bit	512-bit	896-bit	384-bit
Memory Type	DDR	DDR2	GDDR3				GDDR5
Address Space	N/A	No Direct Graphics Memory Access		32-bit			40-bit
	N/A			Single Kernel Execution			Concurrent Kernel Execution
Thread Comm-unication	N/A			Data sharing between local threads			Global thread synch
Memory Hierarchy	N/A	No cache or shared memory		Shared memory and L1 cache		L2 Cache added	Flexible local cache / shared memory
Memory Access-ibility	No Direct Graphics Memory Access			Unified Computational Architecture			Linear Unified Address Space
Error Correction	N/A						Memory ECC Support
Programm-able Data Type Operations	N/A	8-bit pixel ops	Limited floating point capability	Full floating point capability		Limited double precision capability	Full double precision

Table 1 (continued).

	pre 2001	2001	2002-2003	2006		2007
Programm-ability	N/A	Programm-able Pixel Shader (limited)	Larger programs supported	Full C language support to include memory pointers & native data types		C++ support
Program Flexibility	Fixed Functions	Limited Pixel Shading Instruction Set	Larger Pixel Shading Instruction Set	Stream Computing		
APIs	N/A	OpenGL & DirectX 8.0	DirectX 9.0	DirectX 10.0, CTM, CUDA	Stream SDK	OpenCL, CUDA 2.0

In 2007 limited double precision floating point operations were introduced and improved somewhat in 2008 (Chu 2010; Walrath 2008). In addition to shared memory with L1 cache, 2008 brought L2 cache to the HPC researcher. It was not until 2009 that industry saw the realization of the most HPC applicable GPU to date. With the Fermi GPU, NVIDIA introduced support for full double precision floating point operations, linear unified address space with 40-bit addressing support, global thread synchronization and concurrent kernel execution, configurable cache and shared memory, and full Error Correction Code (ECC) memory support (Glaskowsky 2009).

With a linear unified address space and 40-bit addressing available, support for the popular C++ language became possible. Error detection and correction in memory, which is very important for many HPC applications, was not supported for graphics device memory until 2009. Table 1 highlights significant advances in graphics technology that lead not only to GPU

programmability but also to feasibility for utilizing GPUs in HPC applications, including real-time HSI data processing.

Benefits and Limitations

Advances in GPU development with an eye toward general purpose computing have opened the door to opportunities for significantly improved HPC accessibility. Current GPUs have billions of transistors with several multi-processing cores and offer floating point operations in the teraflop range. This high performance is offered by GPU computing at lower cost, lower power requirements, and a smaller footprint than traditional HPC solutions.

While the applicability of GPU computing to HPC research has advanced along with GPU development, GPUs are not yet the true general purpose parallel processing solution provided by supercomputers and computer clusters. GPUs have roots in specialized hardware for graphics acceleration. “Two key attributes of computer graphics computation are data parallelism and independence,” i.e., the same computation is applied to each data element in parallel and each operation is independent of the results of the other operations (Harris 2005, 494). Thus the GPU architecture is optimized for problems that can be framed in a single instruction multiple data (SIMD) model (Owens 2004; Tulloch 2006). Not all HPC computing requirements can be presented in this way and therefore may not achieve the efficiency afforded SIMD problems.

Literature Review

HSI Unmixing and Classification Algorithms

Sergio Sanchez and Antonio Plaza, in collaboration with several different researchers, have produced by far the most literature on the topic of utilizing GPUs for HSI processing. Plaza's primary focus is on pixel unmixing algorithms. Pixel, i.e., spectral, unmixing algorithms work to extract various endmember spectral signatures from a single pixel. Often the spatial resolution of remotely sensed HSI is such that a single pixel may contain a mixture of several different materials. In general, unmixing algorithms assume that the pixel comprises a linear mixture of the endmember materials weighted according to the proportion of the pixel occupied by that material (Sanchez, Martin, Plaza, and Chang 2010). Endmember extraction, or spectral unmixing, divides a pixel into components much like classification algorithms divide an image into classes.

The N-FINDR, Pixel Purity Index (PPI), Orthogonal Subspace Projection (OSP), Linear Spectral Unmixing (LSU), and the Automated Morphological Endmember Extraction [AMEE] algorithms have been implemented on GPUs by various researchers in collaboration with Plaza since GPU computing became available (Plaza, Plaza, and Sanchez 2009; Sanchez and Plaza 2010; Sanchez and Plaza 2011a; Sanchez and Plaza 2011b; Sanchez, Martin, Paz, et al. 2010; Sanchez, Martin, Plaza, and Chang 2010; Setoain et al. 2007). N-FINDR, PPI, and OSP are very popular unmixing techniques; however, none of these unmixing algorithms require calculating a local covariance matrix for each pixel in the HSI scene.

The most computationally intensive aspect of the N-FINDR algorithm Sanchez and Plaza implement requires calculating the determinants of a nonsingular matrix (Sanchez and Plaza 2011a). While this requires the complexity of matrix factorization, Sanchez and Plaza exploit “some basic properties of the LU factorization and matrix determinants” to reduce the computational complexity required of the GPU implementation to achieve a 7.30x speedup over the serial version (Sanchez and Plaza 2011a, 81570F-5). Luo similarly implements the N-FINDR, although achieves a 10.92x speedup with a hybrid solution that employs both the CPU and GPU (Luo 2011).

Yang, Du, and Chen implement the N-FINDR in conjunction with a band selection algorithm to reduce the dimensionality of HSI data and calculate a global covariance matrix in a noise-whitening step. Calculating the matrix determinants is the most computationally demanding calculation of the algorithm, but the bottleneck is in the spatial domain for this band selection application. Therefore, Yang et al. focus the GPU implementation effort on the spatial component of the Band Selection algorithm and realize a 22.67x speedup (Yang and Du 2011; Yang, Du, and Chen 2011). Chang et al. also leverage the power of the GPU to implement a band selection algorithm based on Parallel Particle Swarm Optimization techniques, though this implementation does not require covariance matrix calculations (Chang et al. 2009).

The PPI algorithm Sanchez and Plaza implement consists of calculating the dot product of each pixel vector with a predetermined unit vector. This type of calculation is extremely well suited to the SIMD data model that works best on

the GPU architecture as the 61.28x speedup achieved by Sanchez and Plaza's GPU implementation reflects (Sanchez and Plaza 2010). In an earlier attempt Plaza et al. achieved 26.08x speedup for the PPI algorithm (Plaza, Plaza, and Sanchez 2009).

The OSP algorithm is a deterministic algorithm as opposed to the stochastic algorithms primarily discussed in this paper. The OSP implementation on a GPU Sanchez and Plaza present iteratively performs numerous vector and matrix multiplications in parallel for each pixel in the scene. Sanchez and Plaza (2011b) are able to achieve a 10.65x speedup compared to the 7.3x speedup obtained with the N-FINDR implementation.

Sanchez et al. (2010) implement an LSU algorithm to determine the abundance map for a given set of known endmembers, i.e., spectra expected in the scene. This algorithm requires a matrix-vector multiplication for every pixel in the scene and thus very nicely fits the SIMD model. Sanchez obtains 81.5x speedup with this implementation (Sanchez, Martin, Plaza, et al. 2010). Interestingly, in a later work Sanchez reports a 19.42x speedup for this algorithm (Sanchez and Plaza 2011b). This discrepancy is likely accounted for by the particular hardware configuration for the timing experiments as well as the fact that the later experiment explicitly utilizes only a single core of the four core CPU, while the earlier version may have used more than one core of the eight core CPU in that system.

As with the PPI algorithm Plaza et al. (2009) implement the AMEE algorithm to extract the set of endmembers in the scene. This algorithm extends

“mathematical morphology operators” to find “the most spectrally pure and mostly highly mixed pixel” in a pixel neighborhood (Plaza, Plaza, and Sanchez 2009, 209). To determine the uniqueness of each “spectrally pure” pixel as compared to the “most highly mixed” pixel the spectral angle distance is calculated using the SAM algorithm (Plaza, Plaza, and Sanchez 2009, 209). The GPU implementation of this algorithm gave a 25.57x speedup (Plaza, Plaza, and Sanchez 2009). In earlier, apparently related, work Setoain et al. report that “speedups achieved by the GPU implementation over their CPU counterparts are outstanding: they are in the order of 10” for the AMEE algorithm as described in Plaza (2009) (Setoain et al. 2007, 445).

In the very early days of GPU computing, Plaza collaborated with Setoain et al. to implement the Spectral Information Divergence (SID) classification algorithm. This Automated Morphological Classification algorithm does not require calculation of a covariance matrix. It does, however, employ the concept of a neighborhood window traversing the scene to visit each pixel, similar to the covariance matrix estimation window, though the window in this implementation is only the 3 x 3 neighborhood (Setoain et al. 2006).

More recently Zhang and Lim implement the PCA dimensionality reduction algorithm with NVIDIA’s Fermi architecture, the most GPU computing-friendly GPU available. This implementation shows a 53.37x speedup though the literature is unclear whether a global or local covariance matrix is calculated (Zhang and Lim 2011).

HSI Detection Algorithms

Paz and Plaza introduce a new HSI anomaly detection algorithm and compare the performance with a GPU implementation of the RX algorithm implemented using a global covariance matrix. The new algorithm combines a spatial component with the Spectral Angle Distance algorithm, much like the SAM algorithm, and offers better detection performance than the RX using global statistics. The speedup with the RX algorithm using global statistics is 81.40x and the morphological approach realizes a 17.17x speedup; however, the latter approach takes a total of 4.29 seconds compared to the RX implementation which requires 24.51 seconds for the given data scene (Paz and Plaza 2011).

Winter and Winter implement the PCA and ICA dimensionality reduction algorithms as well as a linear unmixing and the N-FINDR algorithms in their work. They indicate that the RX GPU implementation was “by far the most complex algorithm in the set” (Winter and Winter 2011, 80480O-2). The RX implementation is applied to a scene pre-processed with the PCA algorithm, which reduces the number of bands required for the RX component to process. Winter realizes a 45x speedup with this implementation that requires that a 7× 7 local covariance matrix be calculated for every pixel in the data scene. Speedups for the other HSI algorithms implemented in this work were “on the order of a factor of 10” except for the ICA which showed approximately 3x speedup (Winter and Winter 2011).

Baker et al. were able to get a 3.1x speedup in an effort to implement the Matched Filter algorithm on a GPU when the tools for implementation still

included only pixel shaders. This implementation used a pre-processed covariance matrix calculated based on reference signatures, thus avoiding the requirement for dynamic covariance matrix calculations. This early work clearly demonstrated the need for additional cache in order to reach higher speedups for HSI data processing (Baker, Gokhale, and Tripp 2007).

Tarabalka et al. used the pixel shader APIs available for the NVIDIA GeForce 8800 Ultra to implement an RX-like anomaly detection algorithm “that includes a normal mixture estimation task” (Tarabalka et al. 2008, 990). This implementation of a multi-component statistical model requires calculation of a local covariance matrix, though due to computational costs the covariance matrix is calculated for a subset of the whole image rather than using every pixel in the scene. Results of this work indicate that with the early model programmable GPUs the speedup for these types of calculations were inversely proportionate to the number of bands processed: for 5 bands the speedup was 20x while “a more modest factor of 3 is obtained” when processing 50 bands (Tarabalka et al. 2008, 992).

Recently, Trigueros-Espinosa et al. compare GPU implementations of the RX algorithm with the Matched Filter and an Adaptive Matched Subspace Detector (AMSD). In this work both global and local statistics are used for the covariance matrix calculations to result in speedups of 16.7x and 11.1x, respectively. The Matched Filter speedup achieved was 16.6x and the AMSD speedup 18.4x (Trigueros-Espinosa et al. 2011). Specific GPU implementation details are not given in the literature.

Paz and Plaza implement the RX and OSP algorithms on a GPU in. This implementation appears to utilize global statistics for the covariance matrix calculations. The covariance matrix calculation implementation is unclear in the literature, however, though a speedup of 14.04x and 3.40x is reported for RX and OSP, respectively (Paz and Plaza 2010). Interestingly, Paz and Plaza (2010) give the same OSP algorithm description and use the same test data set though on different GPUs: the first on NVIDIA's GeForce 9800 GX2, which contains two G92 architecture GPUs and the latter on NVIDIA's GeForce GTX 275, which comprises the GT200 architecture. The GTX 275 implementation of the OSP algorithm obtains a reported 51.6x speedup compared to the GeForce 9800 implementation that reaches 3.40x speedup (Paz and Plaza 2010).

Heras et al. implement a Neural Network approach to HSI target detection on the GPU that does not require covariance matrix calculations and sees a 51.7x speedup (Heras et al. 2011).

Other HSI and Related Algorithms

The GPU Computing applications are growing far and wide; as such the literature reveals several related research studies. One such study applies HSI algorithms for authenticating classical works of art (Fresse, Houzet, and Gravier 2010). Although the algorithm used for this authentication does not calculate the covariance matrix for every pixel in the scene, the calculations employed are very similar, calling for an estimation window-like approach. Fresse et al. see an approximate 8x speedup with the GPU implementation of the hyperspectral image comparison algorithm (Fresse, Houzet, and Gravier 2011).

An image classification chain that includes the K-Means algorithm is implemented by Bernabe, Plaza, Marpu, and Benediktsson. This spatial implementation processes an image with a single band and does not require a spectral covariance matrix calculation; however, the GPU solution for the entire chain provided a 36.69x speedup (Bernabe et al. 2012).

The Jacket Toolbox by AccelerEyes is an add-on to the popular MATLAB® programming language. Similarly, the Hyperspectral Image Analysis Toolbox (HIAT) extends MATLAB's functionality by providing "a suite of algorithms for classification and unmixing of hyperspectral and multispectral imagery" (Rosario-Torres and Velez-Reyes 2009, 1). Torres implements a maximum likelihood and Euclidean distance classifiers from HIAT using the Jacket Toolbox and achieves approximately 12x and 17x speedup, respectively (Rosario-Torres and Velez-Reyes 2009).

In other, related HSI work Topping develops an SIMD approach for GPU implementation of nonlinear algorithms, such as the K-nearest neighbor algorithm (Topping 2009). Canty and Nielsen implement an HSI change detection algorithm leveraging the GPULib library by Tech-X Corporation that extends the Interactive Data Language (IDL) for an "order-of-magnitude reduction in processing time" (Canty and Nielsen 2011).

GPU computing solutions for algorithms applied to other types of remotely sensed data are also being explored. Si and Zheng implement a Fourier transform, a Sobel edge detection, and template matching for spatial processing

of 8-bit gray scale imagery, which result in 37.79x, 47.24x, and 6.79x speedups, respectively (Si and Zheng 2010).

CHAPTER III

METHODOLOGY

GPU computing is derived from graphics programming in which hardware optimizations emphasize pixel operations. While this Single Instruction Multiple Data (SIMD) parallelization is very effective for calculating geometric transformations to determine appropriate values for each pixel in a given graphic scene, the specialized hardware designed for this purpose presents a different and somewhat limited perspective for general purpose computing. This chapter describes a methodology for leveraging the GPU computing power for the purpose of processing HSI data. Specifically, the NVIDIA GPU and CUDA architectures are employed to implement the RX algorithm with the intention of speeding up the spectral covariance matrix calculations.

The discussion begins with details for the NVIDIA GPU architecture followed by an in depth discussion of the NVIDIA CUDA API. An explanation of the approach used in this research is then given and includes both serial and GPU-based parallel implementations of the RX algorithm using the Cholesky Decomposition. The experimental setup portrays specifications for the hardware used as well as the parameter configuration used for timing analysis.

Hardware and Software Description

Graphics Processing Hardware

Executing a program on a GPU requires a host system with an independent CPU and the associated components such as motherboard, power

supply, independent dynamic random access memory (DRAM), and a dual slot PCIe, or a single PCIe with room for a double-wide graphics card.

This research utilizes an NVIDIA Quadro FX 4800 GPU, which is in the NVIDIA GT200 architecture series. Thus a detailed description of the GT200 architecture is given. This discussion uses specifications for the most fully featured GPU in the GT200 series, the GTX 280.

Table 3 enumerates specification differences between the GTX 280 and the Quadro FX 4800 and lists G80 architecture differences for context reference.

NVIDIA Terminology

To facilitate a discussion on the NVIDIA GPU hardware it is helpful to first introduce and clarify some NVIDIA terminology that can be misleading at times. GPU hardware comprises processing cores, functional units, and various types of memory. NVIDIA uses the term ‘core’ more loosely than is typical, claiming the GT200 has 240 processing cores. While this is true in one respect, it is not so in the traditional sense. The GT200 has only 30 true processing cores, each with full functionality to include an independent front-end for instruction decoding and a dispatch unit with scheduling logic. Each of these processing cores, or streaming multiprocessors (SM) in NVIDIA terms, contains 8 “streaming processors” (SP) (NVIDIA 2008, 9). These streaming processors are effectively coupled ALUs with minimal logic associated, not fully functional cores with independent front ends and are managed by the streaming multiprocessor; 30 SMs X 8 SPs result in the 240 cores claimed by NVIDIA. These are sometimes referred to as CUDA cores to differentiate from fully functional processing cores.

NVIDIA also overrides the expected definition of a thread. Whereas in traditional usage, a thread implies a relatively costly context switch, in comparison a CUDA thread is “extremely lightweight [with] very little creation overhead” and context switches are “instant” (Ruetsch and Oster 2008, 5). The implementation of the NVIDIA GPU Instruction Set Architecture (ISA) for thread management is realized by what NVIDIA calls a warp, a term borrowed from the concept of weaving threads of cloth on a loom. The GT200 introduces two processing modes associated with this micro-architecture: “graphics processing mode and parallel compute mode” (NVIDIA 2008, 9). A CUDA warp, which is used in the parallel compute mode, contains 32 CUDA threads each of which are designed to execute the same instruction. Warps for the graphics processing mode may contain different numbers of threads (NVIDIA 2008).

The unified architecture NVIDIA first developed with the G80 “was based on a Scalable Processor Array (SPA) framework” and is extended in the GT200 successor (NVIDIA 2008, 9). In addition to the SPs, other functional units on the GPU include texture filtering processors (TF). The term ‘device’ is used to reference the GPU assembly which contains the GPU chip and the associated DRAM.

For the remainder of this discussion the term ‘thread’ will refer to a CUDA thread. The terms ‘core’ and ‘SP’ will be used interchangeably to reference a CUDA core. Other components on the GT200 are referred to by traditional descriptors and are discussed below.

Detailed Hardware Description

The GPU is “specialized for compute-intensive, highly parallel computation … and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control” when compared to CPU designs (NVIDIA 2010, 2). According to NVIDIA, dedicating the majority of transistors to processing mitigates memory access latency with computations rather than with a data cache. Figure 2 illustrates this difference in design. Note that the GPU has more transistors with less cache and flow control than the CPU. This concept is aptly suited for embarrassingly parallel applications such as those used for graphics processing because sophisticated flow control is not necessary when all of the data elements follow the same instruction path (NVIDIA 2010).

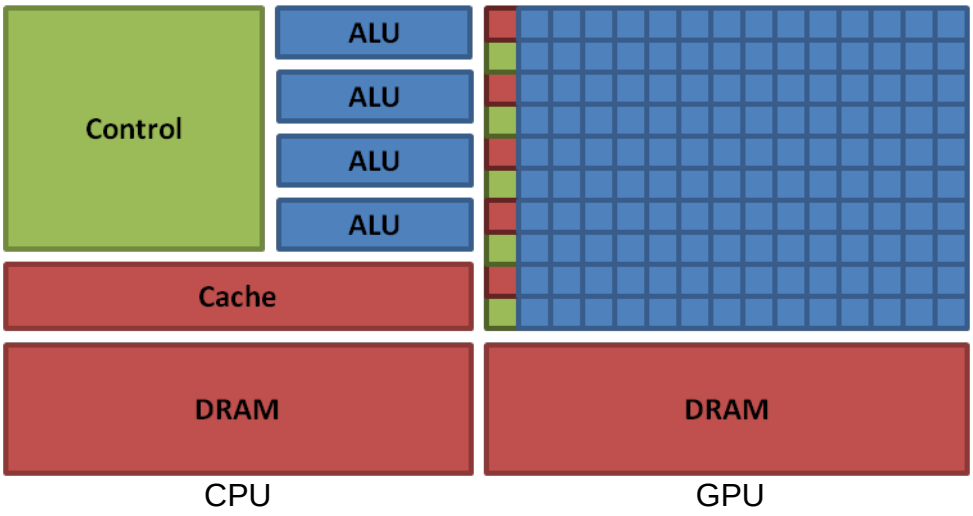

Figure 2. CPU / GPU Design Comparison, diagram derived from (NVIDIA 2010).

Streaming Processors

The NVIDIA GT200 series GPUs, of which the Quadro FX 4800 is a part, comprises a “scalable array of multithreaded Streaming Multiprocessors …

designed to execute hundreds of threads concurrently" (NVIDIA 2010, 77). This GPU design does not optimize "for single-threaded performance as individuals, but for multithreaded performance when operating en masse" (Halfhill 2009, 6). The SPs are the low-level functional units designed to perform the majority of the calculations required of the GPU. Each SP comprises only the pipelined ALUs (Arithmetic Logic Units), both floating point and integer units, instruction and operand dispatcher logic, and a queue for the results, shown in Figure 3.

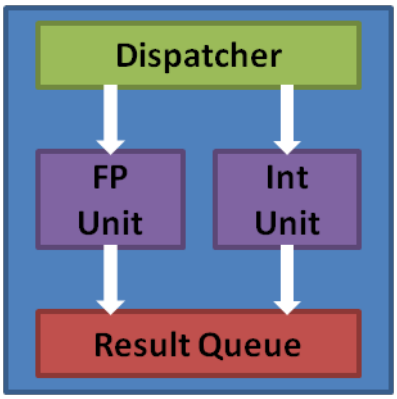

Figure 3. NVIDIA Streaming Processor, diagram derived from (Halfhill 2009).

This simplicity removes the register files, L1 caches, load/store units, and additional floating point and integer units typically found in multiprocessor cores (Halfhill 2009). As such, an SP is optimized to process a single instruction at a time per thread and then quickly switch to another thread (Halfhill 2009).

Streaming Multiprocessors

Each SM has eight SPs and is responsible for creating, managing, scheduling, and executing threads. The SM contains the sophisticated logic for flow control which allows the simplicity of each subordinate SP. In managing warps of threads, the SM holds the execution context for the lifetime of the warp

and thus provides effectively free context switches among threads as mentioned above (NVIDIA 2010). An SM can be working up to 32 warps at the same time, which results in a potential of 1,024 threads (32 threads per warp X 32 warps) “in flight” for each SM (Lal Shimpi and Wilson 2008).

In addition to the SPs and the warp scheduler, each SM contains a double precision floating point unit (DPFPU), two special function units (SFU) for more complex functions such as sine, cosine, and square root, a read-only Constant Cache, Shared Memory, and a significant number (16,384 for compute capability 1.x) of 32-bit registers (NVIDIA 2010). The Constant Cache is shared by all functional units in the SM and is designed to speed up read access to Constant Memory in the Global Memory space which is discussed below. A streaming multiprocessor is depicted in Figure 4.

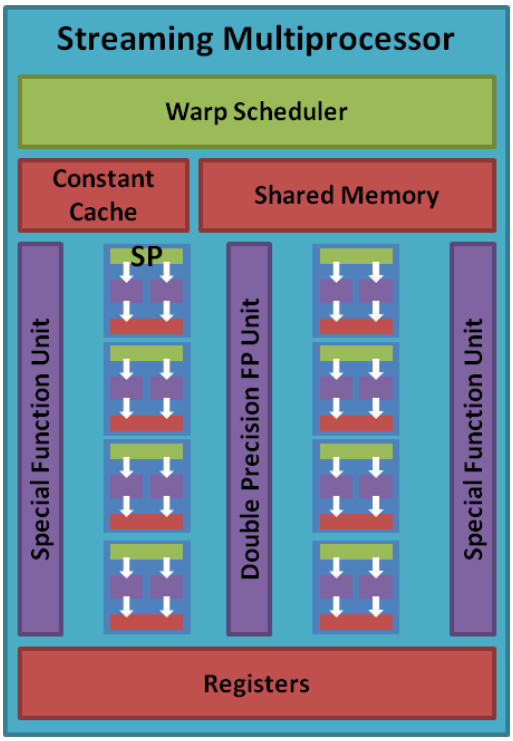

Figure 4. GT200 Streaming Multiprocessor, figure derived from (Abi-Chahla 2008).

Clock cycles required to execute an instruction for all threads in a warp are given in Table 2.

Table 2

Warp Scheduler Instruction Issue Timing for GT200 Architecture

Arithmetic Instruction Description	Clock Cycles Required
Single Precision Floating Point Operation	4
Integer Operation	4
Double Precision Floating Point Operation	32
Single Precision Floating Point Transcendental Operation	16

Thread Processing Clusters

The GT200 architecture continues the modular design at the next higher level by clustering 3 SMs along with control logic for SM scheduling, a texture unit, and a Texture Cache into a Thread Processing Cluster (TPC). The texture units comprise 8 texture addressing units and 8 texture filtering units, shown in Figure 5. The thread addressing units and thread filtering units are indicated with TAU and TFU, respectively (Lal Shimpi and Wilson 2008).

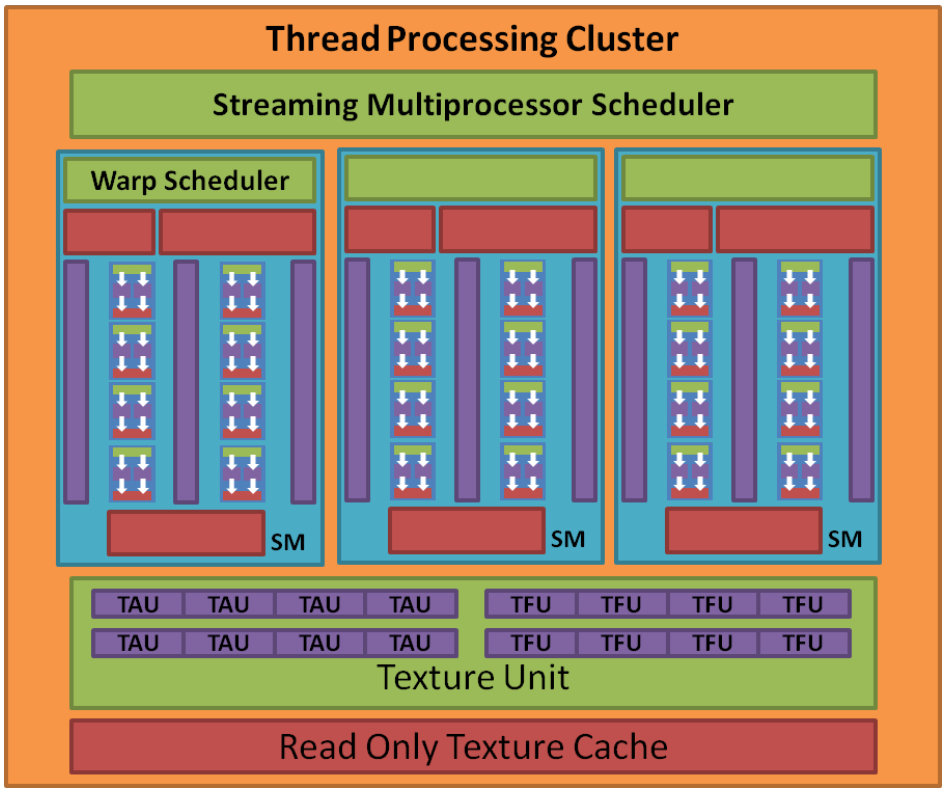

Figure 5. Thread Processing Cluster in the GT200, figure derived from (Lal Shimpi and Wilson 2008).

The Scalable Processor Array mentioned earlier comprises a collection of TPCs. The top of the line GT200 has 10 TPCs, resulting in 30 SMs on the card. This allows support for a theoretical 30,720 threads in flight at the same time. The Quadro FX 4800 used in this research has 8 TPCs for a total of 192 SPs and a theoretical maximum of 24,576 (8 x 3 x 1024) concurrent threads. A top level illustration of the GT200 GPU architecture is shown in Figure 8.

Memory

The concept behind the design of the GT200 architecture is to hide the memory access latency with massive calculation throughput. Memory access, and therefore the memory hierarchy, continues to be an important factor for consideration when coding for the GPU to perform general purpose computing.

The GT200 memory arrangement is significantly different from traditional microprocessors and understanding this different memory architecture is necessary for designing code that executes efficiently.

The GT200 has 8 types of memory in a complex hierarchy: Global Memory, Texture Memory, Constant Memory, Local Memory, Texture Cache, Constant Cache, Shared Memory, and registers shown in Figure 6. Starting from the top, the largest and slowest memory available on the device is the Global Memory and it resides in the DRAM memory that is on-board the graphics card, but not on-board the GPU chip. Special allocation of a portion of the DRAM memory is dedicated as Texture Memory, Constant Memory, and Local (per thread) memory. Accesses to these last three types of memory, while not as fast to access as on-chip memory, are made more efficient through use of caches and registers that are on the chip, as shown in Figure 4 and Figure 5. The Global, Constant, and Texture Memory locations are persistent for the lifetime of the application and therefore provide a means for communication between kernel calls. Kernels are discussed below in the *NVIDIA CUDA Terminology* section below. Local Memory, on the other hand, is valid only through the lifetime of the thread. Similarly, all on-chip memory states exist only through the life of the allocated cluster of threads, or blocks. Blocks are also discussed below.

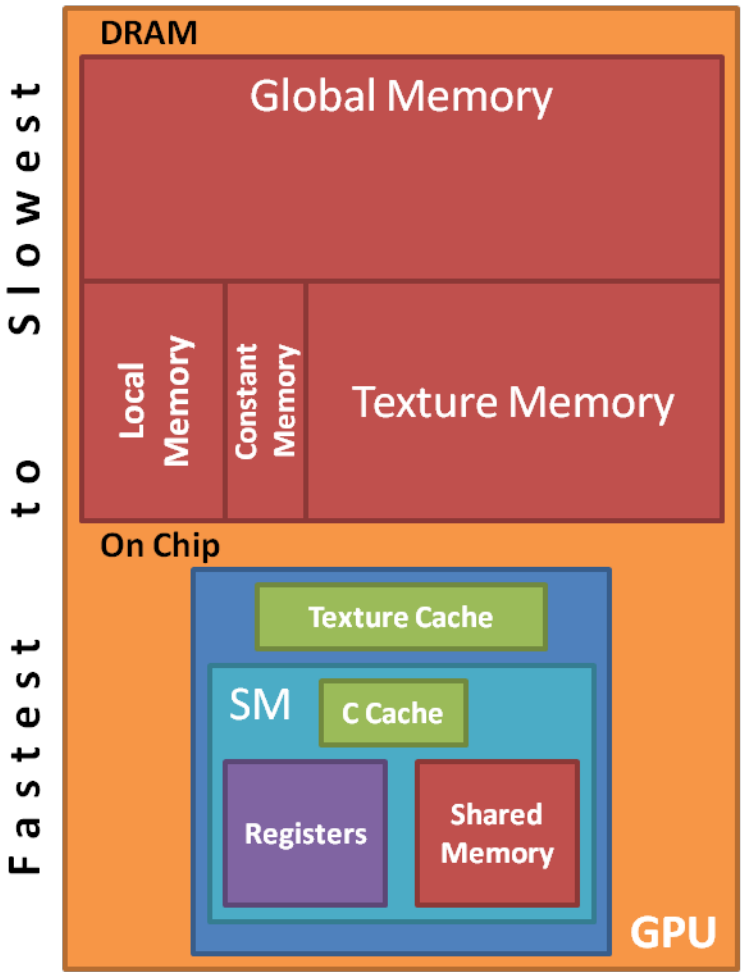

Figure 6. GT200 GPU Memory Architecture.

The fastest memory accesses are to on-chip memory. Registers in the SM are an important fast access memory type, and each SM in the GT200 has 16,384 32-bit registers with which to work (Rys 2008). However, registers are only indirectly accessed by the programmer. Fortunately, the 16KB of Shared Memory on the SMs are easily accessible, almost as fast as the registers and shared by all of the SPs in the SM.

While the Local Memory available for each thread resides off-chip and is un-cached, memory access latency is hidden because the instruction scheduler in the SMs issue fetch and compute instructions independently and

asynchronously (Rys 2007). In this way the registers act as a kind of cache for the thread. The SM also has a true cache for read-only Constant Memory accesses and finally, each TPC has read-only Texture Memory cache for speeding up access to Texture Memory. Figure 7 illustrates the relative memory access speeds and sizes, from the perspective of a thread, for each type of physical memory on the GT200 architecture.

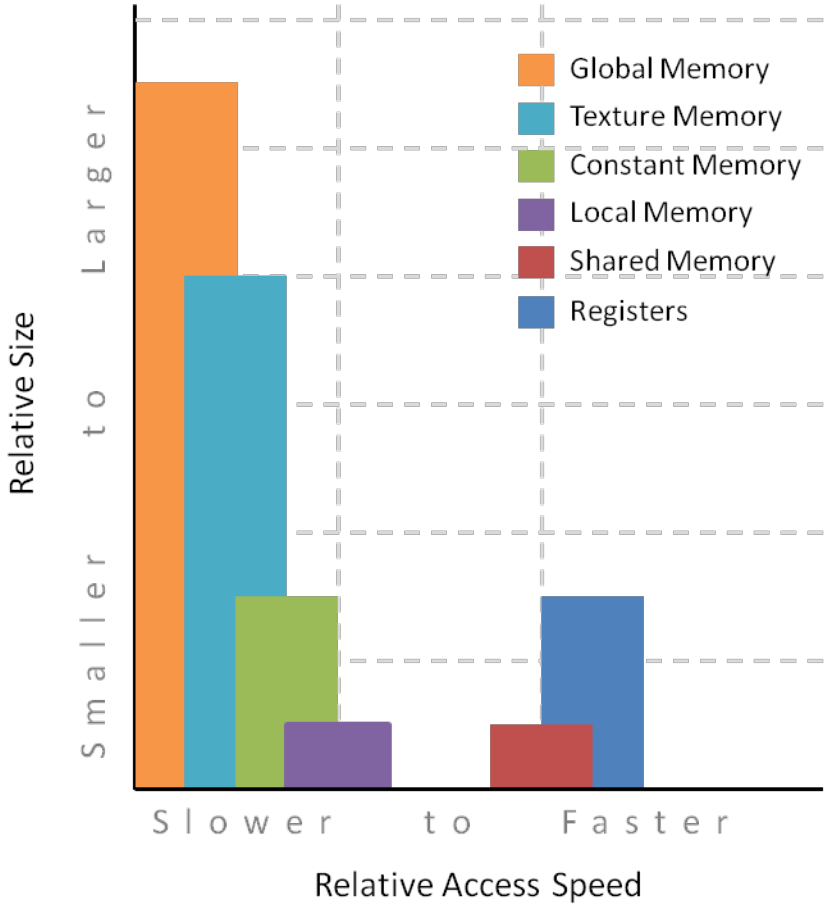

Figure 7. Relative Size and Access Speeds for Different GT200 Memory Types.

Top Level

In addition to the 10 TPCs, the GT200 SPA architecture includes a top level hardware-based scheduler, 8 atomic elements, 8 L2 Texture Caches, and 64B of DRAM. The DRAM is divided into 8 32-bit pairs. Figure 8 offers a high-

level illustration of the GT200 hardware architecture and the *Data Flow* section below describes the instruction and data flow for which this chip is designed.

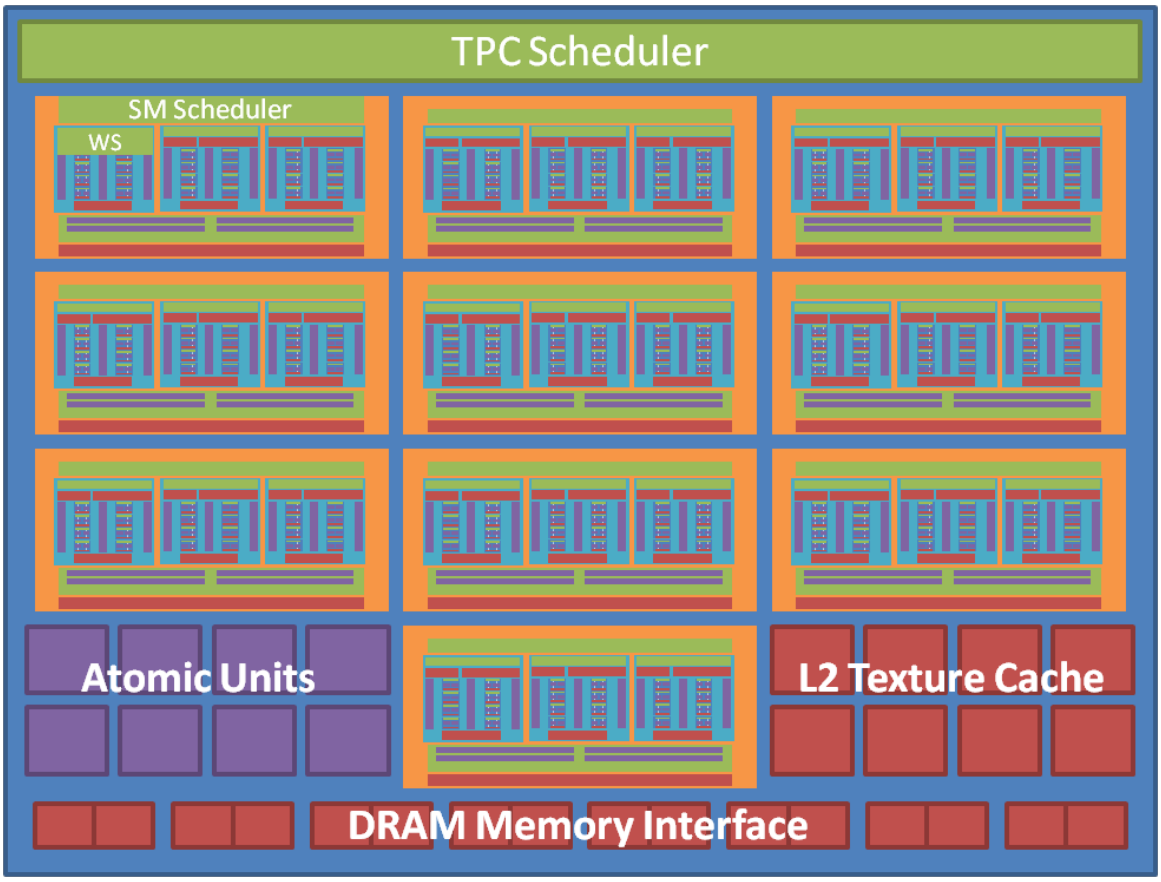

Figure 8. Top Level View of GT200 Architecture, figure derived from (Lal Shimpi and Wilson 2008; NVIDIA 2008).

Specification Comparisons

Table 3, derived from Lal Shimpi and Wilson (2008) and NVIDIA (2008), compares the G80, GT200, and Fermi architecture components relevant to general purpose computing and that may affect speed performance for HSI data processing. Specifications for the Quadro FX 4800 used in this research are also given.

Table 3

Specification Comparison Between the G80, GT200, and Fermi Architectures

Feature	G80	GT200 Quadro FX 4800	GT200 GTX 280	GF100 Fermi
Streaming Multiprocessors per TPC	2	3	3	16 total SMs
Streaming Processors per TPC	16	24	24	32 per SM
Number of TPCs	8	8	10	N/A
Threads per Streaming Multiprocessor	768	1,024	1,024	1,536
Total Threads per Chip	12,288	24,576	30,720	24,576
Total SPs	128	192	240	512
Total Number of Transistors	686 million	1.4 billion	1.4 billion	3.0 billion
Maximum Precision	fp32	fp64	fp64	fp64
PCI Express Bandwidth	6.4GB/s	12.8GB/s	12.8GB/s	12.8GB/s

Compute Unified Device Architecture (CUDA)

The programming interface chosen for this research is the CUDA API developed by NVIDIA. NVIDIA developed this API coincident with developing the G80 architecture which is much like the GT200. In summary, the CUDA API extends the C programming language and has at its core “three key abstractions – a hierarchy of thread groups, shared memories, and barrier synchronization” (NVIDIA 2010, 4). The thread hierarchy establishes a means to nest high resolution “thread parallelism” within a lower resolution “task parallelism” (NVIDIA

2010, 4). The low resolution parallelism consists of sub-problems that can be executed independently while the higher resolution parallelism sub-problems are "solved cooperatively in parallel." This design allows automatic scalability at the coarse-grained level of parallelism because each sub-problem at this level can be assigned to any SP, in any order and can be executed concurrently or sequentially (NVIDIA 2010). A detailed discussion of the CUDA architecture follows.

NVIDIA CUDA Terminology

As with the NVIDIA hardware, NVIDIA has CUDA specific vocabulary worthy of explanation prior to getting into a full discussion of the API design. The "compute capability" of a specific NVIDIA CUDA capable GPU provides nomenclature that coordinates the hardware core architecture with technical specifications and features supported by CUDA. The Quadro FX 4800 has a compute capability of 1.3 which offers capabilities not provided in earlier compute capabilities, such as double precision floating point support. A CUDA kernel is a C function that is called by the host CPU. It resides on the GPU DRAM and executes in parallel once for every thread requested by the function call.

Detailed CUDA Description

One of the most significant concepts to understand in the CUDA design is the thread hierarchy. At the highest level, CUDA offers a logical grid which can be either one- or two-dimensional. The grid comprises logical blocks which can be one, two, or three dimensional. Blocks, in turn, are composed of threads, the lowest level of the thread hierarchy. A one dimensional grid with $D_x$blocks will

have a grid width of D_x. A two dimensional grid with $D_x \times D_y$ blocks will have a grid width of D_x and a grid height of D_y. Similarly, a three dimensional block with $D_x \times D_y \times D_z$ threads will have a block width of D_x, a block height of D_y, and a block depth of D_z. A three component vector serves as a thread index ($threadIdx$) such that $threadIdx.x$ indexes across the block X dimension, $threadIdx.y$ across the block Y dimension, and $threadIdx.z$ across the block Z dimension. A similar paradigm is used for identifying a specific block within the kernel grid (NVIDIA 2010).

Threads are assigned to warps consecutively based on the thread ID. The thread ID can be derived from the thread index (x, y, z) of a block with size (D_x, D_y, D_z) as:

$$threadID = (x + yD_x + zD_xD_y) \quad \text{Eq. 6}$$

(NVIDIA 2010)

This is helpful for designing code that does not diverge within a warp and may improve performance. For instance, given a 32×16 block of threads, the warp scheduler will assign each row to an independent warp. Thus when only two threads are needed for two independent tasks before a synchronization barrier, writing the condition statement so that threads $(0{:}31, 0)$and $(0{:}31,1)$ are given the separate tasks rather than tasking only threads $(0,0)$ and $(1,0)$ is more efficient. The former allows the warps to execute in parallel without divergence. In the latter case, on the other hand, the warp will diverge and execute serially requiring additional clock cycles. Due to the hardware scheduler design discussed above, “branch divergence occurs only within a warp; different warps

execute independently regardless of whether they are executing common or disjoint code paths" (NVIDIA 2010, 78).

The Quadro FX 4800 has compute capability 1.3 for which only one grid may be defined for a kernel and only a single kernel may be invoked at a given time. Figure 9 illustrates the logical configuration options for grid and block layouts available with compute capability 1.3.

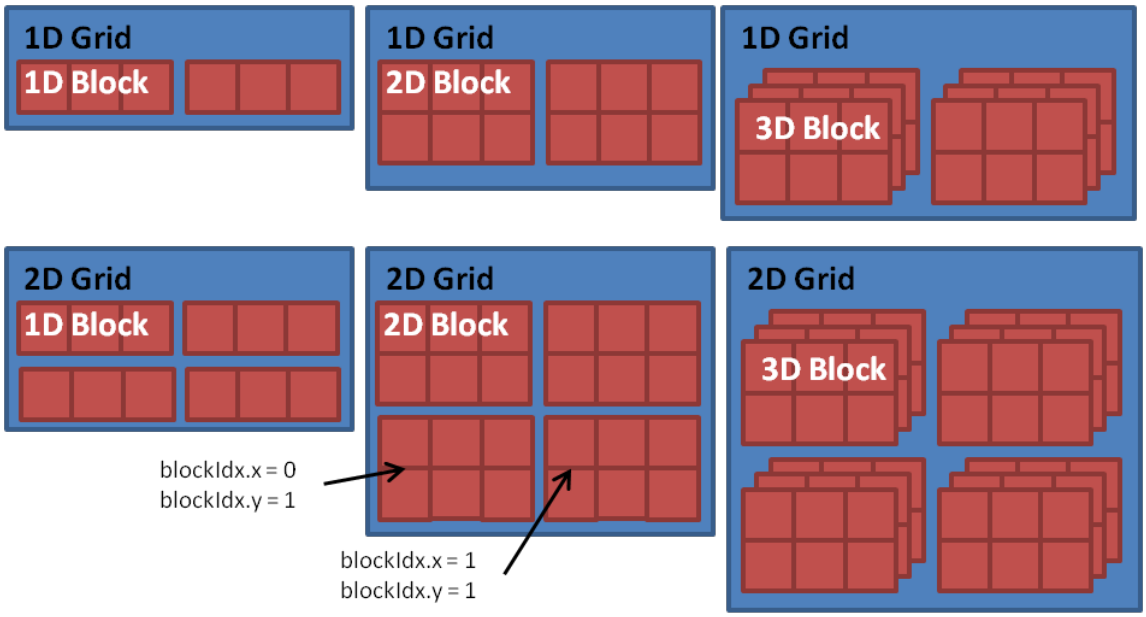

Figure 9. CUDA Compute Capability 1.3 Logical Grid and Block Configuration Options.

Within the blocks, threads may be configured logically in much the same way as the blocks are configured within the grid with the added third dimension; however, block sizes are much more limited than the grid size. See Table 4 for details.

A firm understanding of how CUDA works within the GPU's memory hierarchy is also important for implementing an HSI algorithm to improve speed performance. As mentioned earlier, each thread has read/write (r/w) access to

per-thread Local Memory which resides on the DRAM off-chip. This memory is dynamically allocated for the lifetime of the thread. Additionally, all threads within a thread block have r/w access to per-block Shared Memory which maps to the on-chip Shared Memory in each SM. Finally, all threads within all grids, i.e., for each kernel invocation, have r/w access to Global Memory and read-only access to Texture and Constant Memory off-chip on the DRAM. Figure 10 illustrates these relationships. Memory types are marked as follows: “grid” indicates all threads in the grid have access, “block” indicates all threads in the block have access, and “thread” indicates memory accessed by individual threads i.e. it is not accessible for cooperation with other threads.

A thread synchronization mechanism allows threads within a block to communicate via the per-block Shared Memory and work cooperatively within the block to perform the task assigned to the block. While the Global Memory is persistent across kernel calls, no block synchronization mechanism exists within a kernel. Therefore, blocks within a kernel must be independent. This block independence provides the scalability leveraged by different hardware configurations. Kernels communicate via Global Memory, but they do so in a serial fashion.

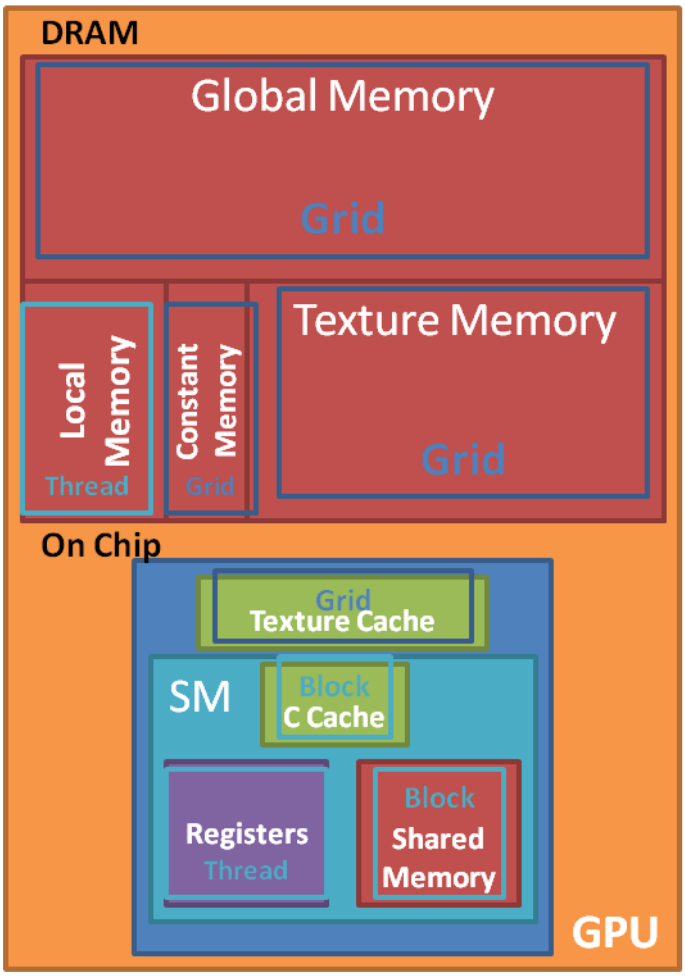

Figure 10. Relationship of Thread and Memory Hierarchies.

Data Flow

Pulling the hardware architecture together with CUDA's software programming paradigm is straightforward. Figure 11 depicts the relationship of the CUDA API architecture to the GPU hardware architecture.

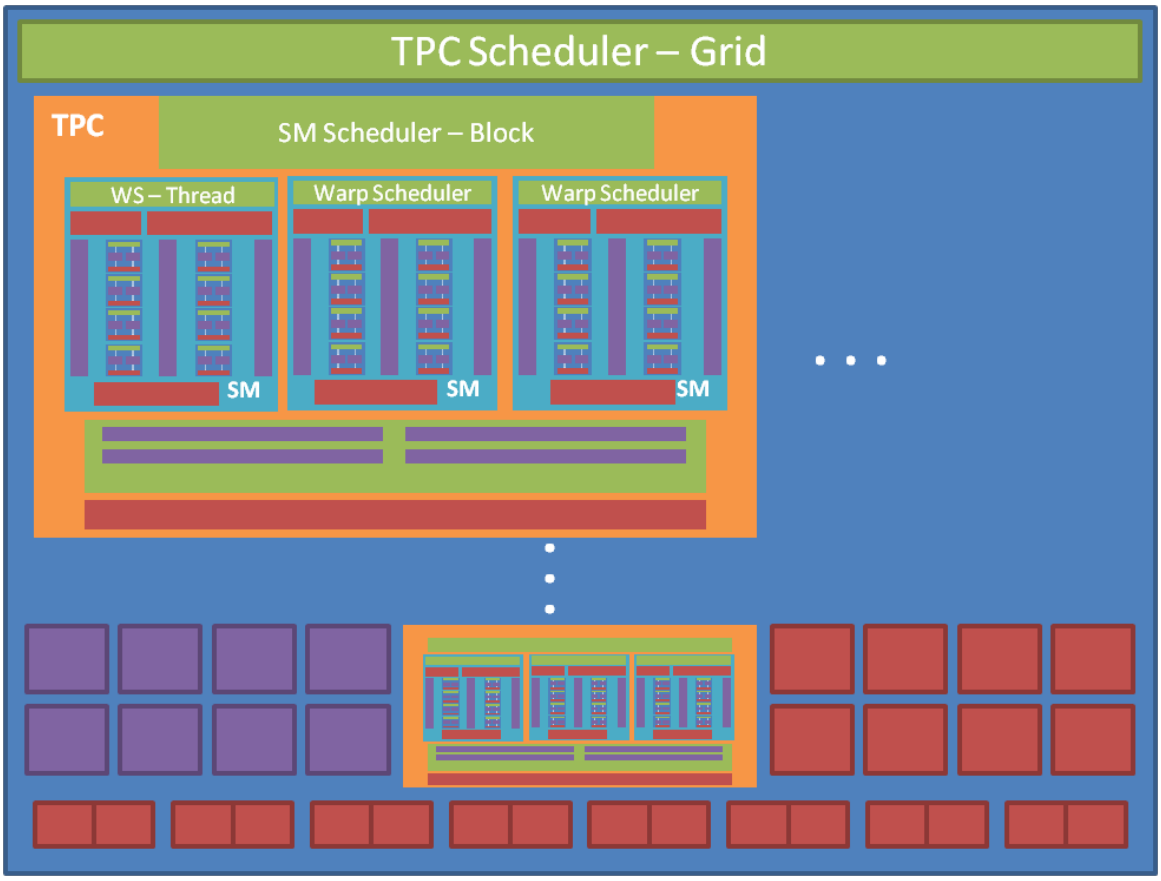

Figure 11. CUDA and GPU Hardware Relationship Diagram, image derived from (Lal Shimpi and Wilson 2008).

To begin the data flow through the system, a program running on the host CPU invokes a kernel to execute on the device sending both data and thread configuration information across the PCIe interface. The kernel also resides on the device for efficient instruction fetches (NVIDIA 2010).

A primary focus of any hardware design effort is to ensure optimal utilization of the available resources. Design of the GT200 is no exception. To that end, sophisticated schedulers are incorporated all along the data flow path beginning with the thread scheduler at the front end of the chip. This hardware-based scheduler works at the CUDA grid level and organizes the CUDA blocks to send to the TPCs. This high-level scheduler offers hardware-based, self load-

balancing and distributes the thread blocks to TPCs as resources become available. The front end also contains logic to manage data sent from the host CPU to the device DRAM. In this way the GPU / CUDA solution offers a multiple instruction multiple data (MIMD) design (NVIDIA 2010).

The TPCs receive several CUDA blocks of threads and the TPC level scheduler feeds the SMs in the cluster as each is free to receive more work. Since the blocks are independent, it does not matter in what order each block is executed (NVIDIA 2010).

The last scheduler in the path is at the SM level. The SM thread scheduler, or warp scheduler, pipelines the thread instructions for all warps. It is via this pipeline scheduling that the SM can have 32 warps in flight concurrently, resulting in the aforementioned 1,024 concurrent threads per SM. All threads in a warp execute the same instruction. The simple SPs do not have branching logic so if a branch occurs within a warp all threads in that warp follow all instruction paths and the SM chooses the correct result for each thread at the other end. This effectively serializes that warp until the threads are brought together again by the logic in the SM. Thus, at this level of the architecture the GPU / CUDA solution presents a single instruction multiple data (SIMD) model (NVIDIA 2010).

The SM has only a single DPFPU which requires 32 clock cycles. For most efficient utilization of the SM resources it is best to work with integer or floating point operations where possible. The DPFPU is rarely needed for typical graphics processing and is only included on the chip for GPGPU computing, which is currently a minority of users. Nevertheless, with sophisticated pipelining

schedulers and 30 SMs, double precision operations are effectively parallelized on the chip, if not as massively as floating point operations (NVIDIA 2010).

At this point in the data flow, memory access latency may become an issue if the register pressure is high enough to cause the scheduler to back off on the asynchronous data fetches. Careful consideration of register utilization is therefore important for efficient code execution. Utilizing Constant and Texture Memory can provide faster read access to memory than Global Memory reads. The TPC Texture Cache can provide faster Texture Memory reads and the SM Constant Memory cache can provide even faster Constant Memory reads. Both of these caches are read-only, however, so in the event Shared Memory is insufficient for the needs of a computation and an intermediate write is required to Global Memory the calculation will likely take a latency penalty (NVIDIA 2010).

After the threads in a block complete the assigned tasks, the calculated result is written to Global Memory and the life of that block ends. A single write to Global Memory for the final result offers the optimal efficiency in terms of memory access. When all blocks in a kernel complete the assigned tasks and all results are written to Global Memory on the device, DRAM control is returned to the host CPU. At this point the host can either retrieve the results from the device via the PCIe bus or invoke another kernel which can utilize the intermediate results left in Global Memory on the device and repeat the data flow process (NVIDIA 2010).

Approach

The following discussion elucidates the approach taken to establish a baseline analysis of the GPU performance for processing HSI data with the RX

algorithm. A serial solution is implemented and discussed followed by details of the GPU solution. The method for dividing the problem in light of the GPU architecture is given, followed by implementation details and a discourse on the tradeoffs inherent to the solution. This chapter is concluded with a description of the experimental setup, which includes a description of the sample data utilized and additional hardware specifications.

Serial Solution

Before beginning the design of a GPU-based parallel solution for HSI processing algorithms, research with several serial solutions is conducted. The following objectives motivate the serial effort of this research:

- Determine a baseline for timing comparisons
- Verify the covariance matrix calculation is the bottleneck of the algorithm
- Provide a truth result for accuracy comparisons
- Determine which linear algebra technique to use for solving the RX equation.

Recall Eq. 4 describing the RX algorithm.

$$RX(\mathbf{p}) = \left(\mathbf{p} - \boldsymbol{\mu}\right)^T \boldsymbol{\Sigma}^{-1}\left(\mathbf{p} - \boldsymbol{\mu}\right) \qquad \begin{cases} no\ target, RX(\mathbf{p}) < t \\ target, \quad RX(\mathbf{p}) \geq t \end{cases} \quad \text{Eq. 4}$$

Rewriting this equation as

$$RX(\mathbf{p}) = (\mathbf{b})^T \mathbf{A}^{-1}(\mathbf{b}) \qquad \begin{cases} no\ target, RX(\mathbf{p}) < t \\ target, \quad RX(\mathbf{p}) \geq t \end{cases} \quad \text{Eq. 7}$$

and setting $x = \mathbf{A}^{-1}\mathbf{b}$ it is easy to see part of this equation has the form $\mathbf{A}x = \mathbf{b}$.

Options explored to calculate the RX result for a given pixel are Gaussian Elimination which requires inversion of the covariance matrix, Cholesky Decomposition which takes advantage of the fact that the covariance matrix is positive semi-definite, and the iterative Gauss Seidel technique (Press et al. 1992, 674). Serial implementation of these three methods reveals that the iterative Gauss Seidel provides different detection results than either the Gaussian Elimination or Cholesky Decomposition implementations. Both Cholesky Decomposition and Gaussian Elimination techniques give similar and expected results. Additionally, analysis of the serial implementation verifies that the most significant bottleneck in the algorithm is the covariance matrix calculation. Based on these results of the serial implementation and to avoid matrix inversion, the Cholesky Decomposition technique is chosen to implement for the parallelized GPU solution.

The pseudo code for the serial program is listed below. The data structure chosen to ensure contiguous memory access is a one-dimensional array. Code for the Cholesky Decomposition is taken from (Press et al. 1992, 96-98).

```
1   Get Input
2   Read data into 1D array
3   For Each Pixel
4       // Calculate Band x Band Covariance Matrix
5   Initialize sum vector and covariance matrix
6   For all band pairs
7   For all pixels in the local estimation window
8   Sum pixels within band
9   Sum band pair product
10  Calculate mean vector from the sum vector
11  Demean sum of product for each band pair in upper diagonal of
    covariance matrix
12  Copy upper diagonal to lower diagonal
13      // Calculate Result
14  Subtract mean vector from pixel vector
15  Solve Ax=b for x using Cholesky Decomposition (where A is the
    covariance matrix and b is the demeaned pixel)
16  Perform Vector Vector Multiply between resulting vector x and demeaned
    pixel vector
17      Enter resulting scalar value into output image at current pixel location
18  Write output image to file
```

Pseudocode 1. Serial Program Design Using Cholesky Decomposition.

GPU Solution Space

Decisions for parallelizing an algorithm to be implemented on a GPU are necessarily different than decisions for traditional HPC parallel systems. Specifically, the GPU has a higher processor-to-memory ratio than a comparable HPC system. Thus, due to Global Memory access latency it may be faster to repeat a calculation than to calculate it once and store it to Global Memory for re-use as is common in traditional HPC solutions (NVIDIA 2009b). A detailed discussion of the considerations necessary for parallelizing the RX algorithm on the GT200 GPU architecture using the CUDA API is given below. Implementation details are then given, followed by a discussion of the associated tradeoffs.

Dividing the Problem

Parallelization of any algorithm onto any architecture requires a solution that divides the problem into tasks that can be performed concurrently. This in turn necessitates a clear view of the task dependencies. Additionally, considerations of the specific hardware architecture are necessary for effective and efficient use of the available resources. Finally, a thorough understanding of the limitations of the hardware interface ensures a design that is implementable with the available programming interface.

Task Dependency Considerations

In preparation for designing a parallel solution of the RX algorithm, a task dependency and interaction graph is created. Figure 12 names the major tasks required to perform the RX algorithm for a single pixel and illustrates the dependencies found among the tasks. The notation indicates relative storage and time requirements for each task. This graph is a useful tool for dividing the problem into blocks that can be executed independently.

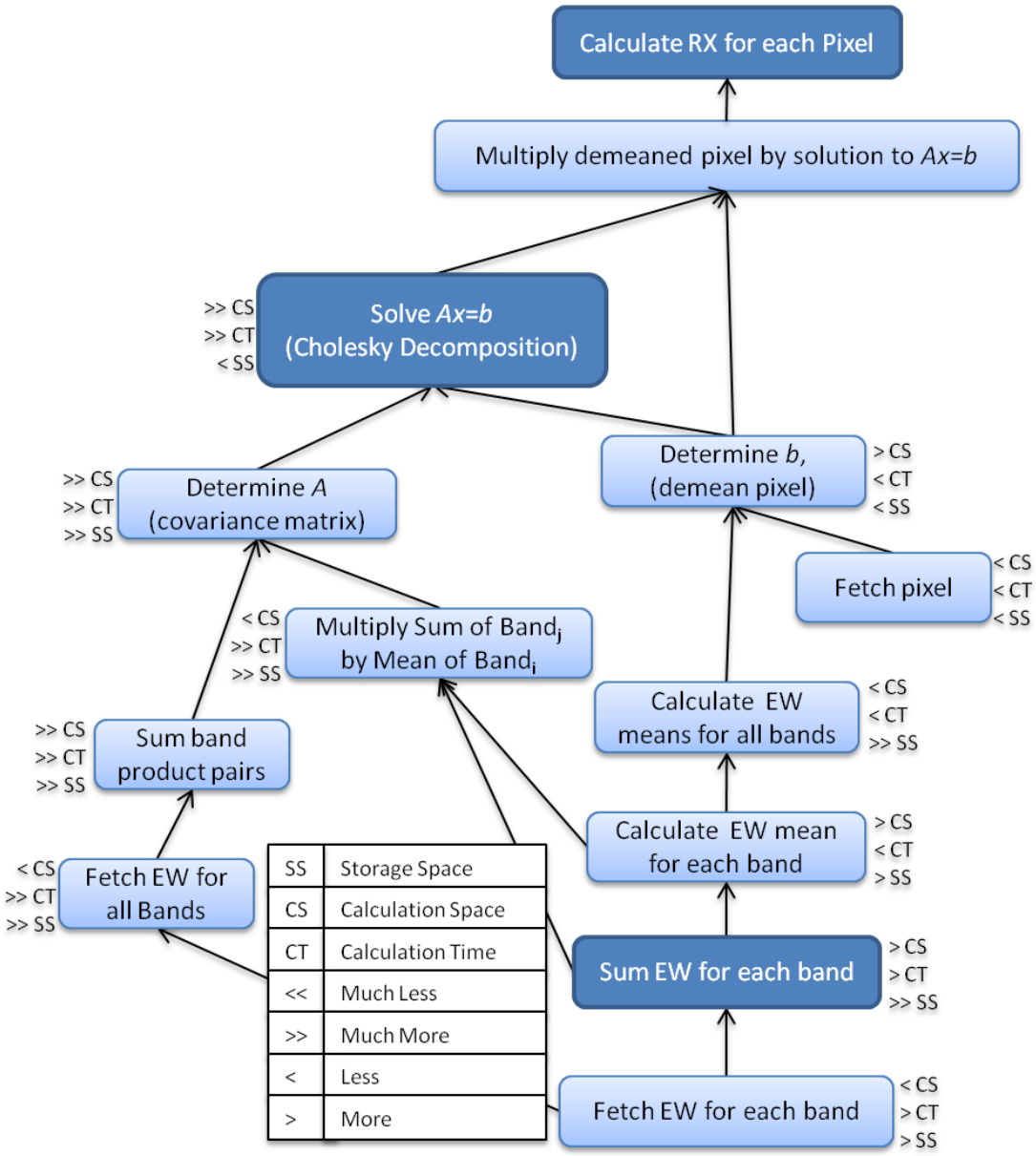

Figure 12. RX Dependency and Interaction Graph.

Calculating a local spectral covariance matrix for every pixel in the data scene is by far the most significant bottleneck of the algorithm. Recall the spectral covariance between band i (b_i) and band j (b_j) at any given pixel is estimated as

$$cov(b_i, b_j) = (1/N)\sum_{k=1}^{N}(b_{i_k} - \mu_i)(b_{j_k} - \mu_j) \quad \text{Eq. 8}$$

where N is the number of pixels in the Estimation Window (EW) and μ_i and μ_j are the EW averages for bands i and j, respectively.

Expanding this we see

$$cov(b_i, b_j) = (1/N)\sum_{k=1}^{N}(b_{i_k}b_{j_k} - b_{i_k}\mu_j - b_{j_k}\mu_i + \mu_i\mu_j) \quad \text{Eq. 9}$$

$$cov(b_i, b_j) = (1/N)\left(\sum_{k=1}^{N} b_{i_k}b_{j_k} - \sum_{k=1}^{N} b_{i_k}\mu_j - \sum_{k=1}^{N} b_{j_k}\mu_i + \sum_{k=1}^{N} \mu_i\,\mu_j\right) \quad \text{Eq. 10}$$

$$cov(b_i, b_j) = \frac{\sum b_{i_k}b_{j_k}}{N} - \frac{\sum b_{i_k}\mu_j}{N} - \frac{\sum b_{j_k}\mu_i}{N} + \frac{\sum \mu_i\mu_j}{N} \quad \text{Eq. 11}$$

$$cov(b_i, b_j) = \frac{\sum b_{i_k}b_{j_k}}{N} - \frac{\sum b_{i_k}}{N} * \mu_j - \frac{\sum b_{j_k}\mu_i}{N} + \frac{N\mu_i\mu_j}{N} \quad \text{Eq. 12}$$

$$cov(b_i, b_j) = \frac{\sum b_{i_k}b_{j_k}}{N} - \frac{\sum b_{i_k}}{N} * \mu_j - \frac{\sum b_{j_k}\mu_i}{N} + \mu_i\mu_j \quad \text{Eq. 13}$$

$$cov(b_i, b_j) = \frac{\sum b_{i_k}b_{j_k}}{N} - \mu_i * \mu_j - \frac{\sum b_{j_k}\mu_i}{N} + \mu_i\mu_j \quad \text{Eq. 14}$$

$$cov(b_i, b_j) = (1/N)\left(\sum_{k=1}^{N} b_{i_k}b_{j_k} - \sum_{k=1}^{N} b_{j_k}\mu_i\right) \quad \text{Eq. 15}$$

$$cov(b_i, b_j) = (1/N)\left(\sum_{k=1}^{N} b_{i_k}b_{j_k} - \mu_i\sum_{k=1}^{N} b_{j_k}\right) \quad \text{Eq. 16}$$

This calculation is required for all band combinations for every pixel in the scene. The 512×512×32 scene used in this research requires this calculation to be performed 1,024 times for each pixel, or 528 times with 496 copies, to fill out the 32×32 symmetric covariance matrix. That is a total of 268,435,456

computations to determine the covariance matrix for all pixels in the scene. Furthermore, each of these computations is dependent on at least $2N$ trips to memory to retrieve data values. The EW size for this study is 33×33; thus a brute force approach requires a minimum 292,326,211,584 trips to memory in addition to the required computations. Figure 12 describes the dependencies necessary for this calculation.

The most straightforward calculation of the RX result of a single pixel calls for both the full covariance matrix and full estimation window to fit into the fastest available memory. For the representative HSI data this requires 32×32×8=8KB for the covariance matrix of double precision floating point values and 33×33×32×2=68KB for all bands of 2 byte integer data in the estimation window. Other smaller memory requirements for temporary variables are also needed.

Hardware Design Considerations

In general the hardware bottleneck is associated with memory accesses and data transfer rather than computations. While very fast relative to past data buses, the PCIe bus from the host CPU to the device DRAM is extremely slow relative to on-device memory buses. It is therefore prudent to limit the data transfer between the CPU and device whenever possible.

Similarly, as seen from the hardware description given above, the access latency to Global Memory is slower than access to the Shared Memory on each SM. However, this Shared Memory is insufficient to contain all of the data needed for calculation of the covariance matrix. Even if the solution took into consideration the symmetric nature of the covariance matrix, sufficient memory is

unavailable. Thus, holding the entire estimation window local to an SM is not an option. Similarly, Constant Memory on DRAM at only 64KB, although faster than Global Memory due to cache availability, is insufficient to hold all of the input data so that it is available for every pixel concurrently. Texture Memory then is the fastest memory, due to caching capabilities at the TPC level, available with sufficient space that will allow concurrent access to all processors of the input data required for each estimation window. Fortunately, the design of the GT200 Texture Memory is optimized for “spatial locality in cached Texture Memory, ” which will benefit access to the spatial EW of each band for each pixel (NVIDIA 2009b, 4).

Programming Interface Considerations

The single kernel allowed to be invoked at any given time by compute capability 1.3 prevents communication between thread blocks and forces an effective “block sync” between kernel calls. Block level parameters do not outlive the block and therefore anything needed by the subsequent kernels must be stored in Global Memory which persists between kernel calls.

The CUDA API offers relatively low level access to the GPU hardware. It also, however, imposes various limitations in an effort to guide the programmer to optimal use of the hardware design. These limitations are seen in the grid and block size ranges as in the limits on the number of threads a block may be assigned. The *Features and Technical Specifications for Compute Capability 1.3* section below enumerates relevant limits established by the CUDA API for the Quadro FX 4800.

Thread-level local variables reside in registers unless the variable is an array or register pressure is too high in which case the variable resides in Local Memory on the DRAM (CVG 2011). Thus, careful consideration of register pressure will facilitate efficient variable access at the thread level.

Features and Technical Specifications for Compute Capability 1.3

Features provided with Compute Capability 1.3 that are not supported for earlier Compute Capabilities are listed here directly from (NVIDIA 2010):

- Integer atomic functions operating on 64-bit words in Global Memory
- Integer atomic functions operating on 32-bit words in Shared Memory
- Double precision floating point numbers

Other atomic operations, including floating point atomic addition are not supported for compute capability 1.3.

Technical specifications of the Quadro FX 4800 relevant to HSI processing algorithm implementation are given here, directly from (NVIDIA 2010).

Table 4

Quadro FX 4800 Technical Specifications (NVIDIA 2010)

Description	Specification
Maximum x- or y-dimension of a grid of thread blocks	65535
Maximum number of threads per block	512
Maximum x- or y-dimension of a block	512
Maximum z-dimension of a block	64

Table 4 (continued).

Description	Specification
Maximum number of resident blocks per SM	8
Maximum number of resident warps per SM	32
Maximum number of resident threads per SM	1024
Number of 32-bit registers per SM	16k
Maximum amount of Shared Memory per SM	16KB
Amount of Local Memory per thread	16KB
Constant Memory size	64KB
Cache working set per SM for Constant Memory	8KB
Cache working set per SM for Texture Memory	8KB
Maximum width for a 1D texture reference bound to a CUDA array	8192
Maximum width for a 1D texture reference bound to linear memory	2^{27}
Maximum width and height for a 2D texture reference bound to linear memory or CUDA array	65536 x 32768
Maximum number of instructions per kernel	2 million

Implementation Details

Implementation considerations include deciding the CUDA grid / block layout and the specifics for parallelizing the algorithm. These considerations are discussed below. Pseudo code is given to help describe the parallelization algorithm.

Grid / Block Layout

This section discusses the chosen logical grid and block layout implemented. The layout design is derived with consideration for warp scheduling, memory access latency, and block size limitations. Alternative considerations for use with the other CUDA compute capabilities are indicated when a decision is specific to compute capability 1.3 supported by the Quadro FX 4800.

Warp Scheduling

A key consideration for choosing the design of the grid and block layout is the warp size and how the warp scheduler assigns instructions for the threads in the warp. As Eq. 6 shows, the warp scheduler assigns threads to execute across the x dimension before the y dimension, which in turn is executed before the z dimension. Given a block width that is the size of a warp, the entire block row is designated to execute the instruction before the first element of the next block row is designated, and so on for the z dimension. With this in mind, divergence in the warp is minimized by allocating tasks at least at the half warp granularity where possible. The warp scheduler assigns threads to fetch from memory at the half warp resolution. All compute capabilities have the same warp logic. It is optimal to have the block dimensions divisible by the size of a half warp.

Memory Access Latency

Considerations for memory access latency are: the register availability; size of Shared Memory; appropriate use of Constant and Texture Memory caches; and the understanding that, while relatively slow when compared with

these faster memory types, Global Memory is on-board the GPU and has DDRAM3 access speeds, which is faster than traveling back and forth to DRAM on the host.

Block Size Limitations

Considering the size of a warp and the thread scheduling logic in the warp scheduler, it is natural to arrange the blocks with an x dimension that is divisible by 32, or 16 as the first dimension designated. The decision to designate threads in the y dimension is governed by the decision of how many threads are assigned in the x dimension; similarly the size of the z block dimension is dependent upon how many total threads have already been assigned. Assigning dimensions in the order x, y, z is a natural choice considering the warp scheduler logic.

Block size limitations given in (NVIDIA 2010) that influence algorithm implementation design decisions are shown in Table 5.

Table 5

Block size limitations

Compute Capability	Total Threads Per Block	x or y Dimension of Block	z Dimension of Block
1.x	512	512	64
2.0	1024	1024	64

Data Format

The organization of HSI data varies depending on the sensor system that collected the data and any pre-processing that may have been applied to the data prior to being presented for detection processing. Typical data formats include band interleaved by pixel (BIP), band interleaved by line (BIL), and band sequential (BSQ). Differences in these formats are illustrated in Figure 13. The type of sensor used for data collection dictates the most efficient storage format for the raw data. Scanning sensors store data most efficiently in BIP format while push-broom systems more naturally have a BIL format. A staring sensor that captures an entire frame per band at a given instance efficiently stores data in the BSQ format.

In designing an algorithm implementation that functions efficiently with data in the raw format, it is necessary to consider the ramifications of the input format. How will the data be efficiently delivered to the GPU, stored on the device, and accessed by the multiple processors? What access patterns are efficient for the necessary computations? Decisions with respect to these questions and relative to this research are given below.

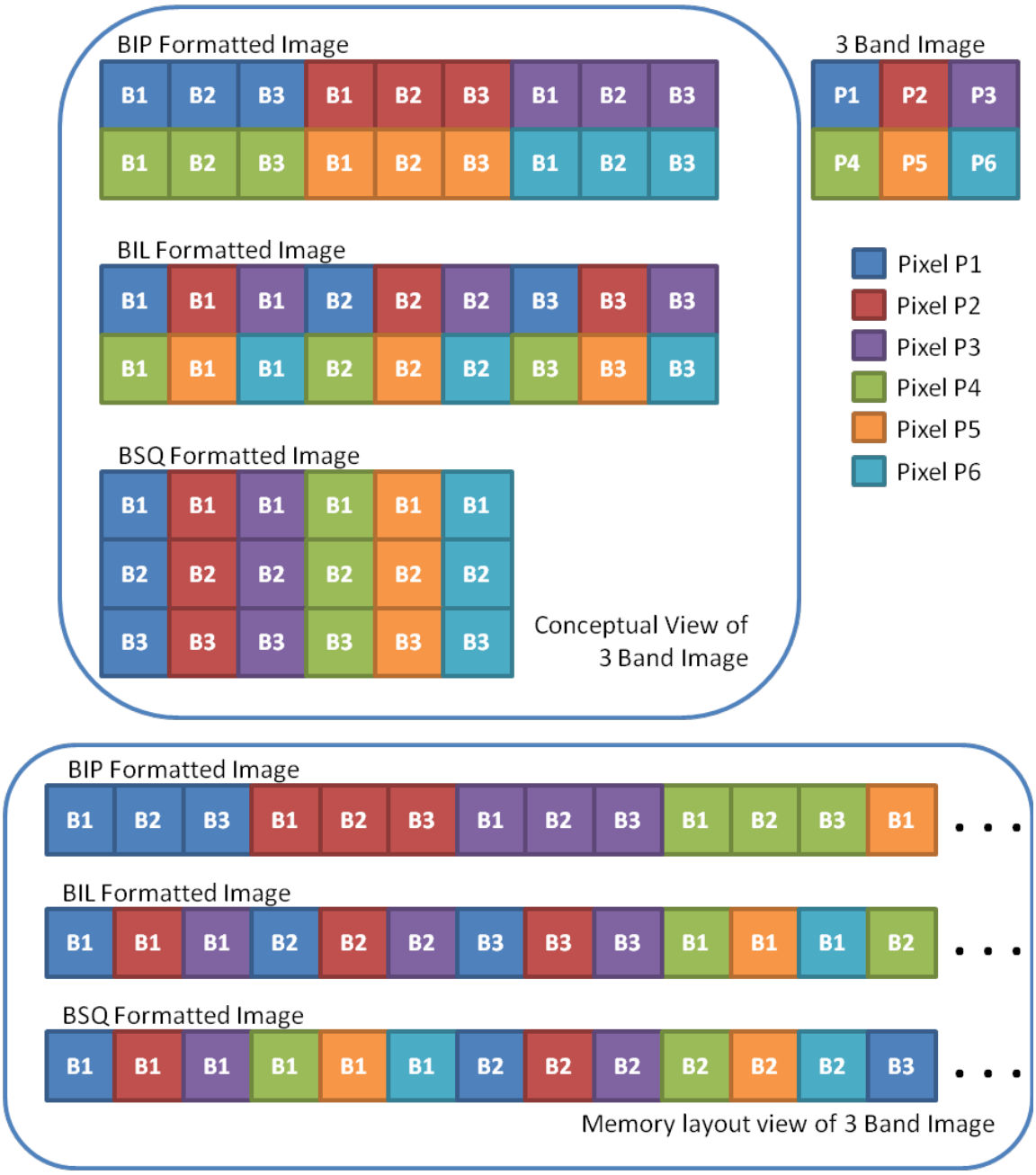

Figure 13. Illustration of BIP, BIL, and BSQ Data Formats.

RX Algorithm Parallelization

With the above considerations for choosing a grid / block design in mind, this section discusses how this design is used to parallelize the RX algorithm,

with particular emphasis on the parallelization of the covariance matrix calculation.

An important consideration for calculating the covariance matrix is the availability of double precision floating point units (FPUs). All SPs have FPUs so that whenever single precision is sufficient, variables should be typed as *float*. Additionally, single precision operations require fewer clock cycles than do double precision operations. For this algorithm, however, single precision operations yield inaccurate results when calculating the covariance matrix. Therefore, double precision operations are needed. While the hardware availability of DPFPUs does not directly modify the design decisions for using the CUDA API, a difference in availability will offer significant performance differences. The G80 architecture supports double precision calculations with the SFUs in the SM while the GT200 architecture offers a dedicated DPFPU in the SM (Bolotoff 2010). Beyond this, the GF100 “Fermi architecture has been specifically designed to offer … up to 16 double precision fused multiply-add operations … per SM, per clock…” (NVIDIA 2009a, 9). The Fermi solution replaces the FPUs in the SPs of the SM with FPUs that perform double precision as well as single precision operations; therefore, timing on a Fermi system will yield faster double precision results than on the GT200 for concurrent calculations. The CUDA code design need not change between compute capabilities in order to realize these speed improvements.

On the other hand, the design decisions elaborated on below reflect accommodations in the CUDA code design necessary for implementation on

different hardware architectures. When appropriate these are indicated and alternate considerations for other compute capable solutions are given.

Beyond the structures imposed by the hardware and API constraints, the grid / block layout is driven by methods for breaking the RX algorithm into parallelized components. This research divides the RX algorithm into three components driven by the information gleaned from the interaction and dependency graph shown above in Figure 12. As shown in the graph, the most time consuming computations of the algorithm involve repeated trips to memory to get data from the EW; first in calculating the average across the EW for each band and then for calculating the sums of the band pair products across the EW. Given sufficient capacity to hold the EW resident in the SM Shared Memory, these calculations could be extremely efficient. However, the current SM Shared Memory limitations are not sufficient to keep resident an EW of the size often chosen for use with this algorithm. Therefore, a solution that supports off-chip EW access is given.

The first computational component for calculating the RX algorithm is calculating the sum that will be used to derive the mean of the EW for each band; the second component is calculating the sums of the band pair products across the EW; and the last step brings the first two terms together and finalizes the RX calculations. These three components, highlighted in Figure 12, are divided into two CUDA kernels, each with a different grid / block layout. The last two steps are completed in the same kernel using the same grid layout. Details are discussed below.

Band Sum of Estimation Window

While the computational capacity on the GPU is significant it is clear from the discussion above that fast storage is limited. Thus, recalculating a result may at times be more efficient than calculating it once and storing it in Global Memory for retrieval later as is often the choice in traditional HPC solutions. Nevertheless, due to the numerous trips to Global Memory required to calculate the sum for an EW band, recalculating this sum is not more efficient than calculating the sum once and retrieving the single sum value from Global Memory as it is needed again. When calculating the covariance matrix for a single pixel of interest, the band sum of the EW for any given band is required $2B$ times, where B is the number of bands in the data set. Therefore, although a trip to Global Memory is slow, it is much more efficient than recalculating the sum, especially as the EW or B increases.

This EW sum is calculated in a CUDA kernel with a grid layout as shown in Table 6. The results of the band sum across the EW of the pixel of interest is then stored in a global device memory scratch pad allocated to be the same size and “shape” as the input data. This requires that the DRAM have sufficient memory for double the size of the input data. Considering the current GPU DRAM capacities and the typical HSI image this is not unreasonable. In the event that the DRAM capacity is insufficient, this solution may be divided as necessary so the host can call each kernel in a loop to accommodate the division. This will combine a serialized approach with the parallelized solution based on the available resources.

Table 6

Grid Layout for EW Sum Kernel

Logical Component	Dimension Size	Index Relationship
Grid Width	Num Data Bands (*numBands*)	Data Band – one block in grid row per band
Grid Height	Num Data Rows (*dataHeight*)	Data Row – one grid row per data row
Block Width	nearest power of $2 \leq \sqrt{N}$ where N is the size of the EW	Estimation Window Row
Block Height	Chosen Total Block Threads / Block Width where Chosen Total Block Threads $\leq$ Max threads	Data Row Subsection – number of subsections the row is divided into to provide "Block Height" way parallelization allocates up to the maximum number of threads

The logical association for this layout is that each block calculates the EW sums for a single band for all pixels in a row and stores the results in the global scratch pad. Thus $blockIdx.x$ and $blockIdx.y$ identify the band and row for the block, respectively. The block width allocates threads across the x dimension in such a way that the block width size is the nearest power of two that is less than the square root of the EW size. The size of the EW is defined as $\sqrt{N} \times \sqrt{N}$ and $ew_d = \sqrt{N}$ where $ew_d$is odd. The $threadIdx.x$ index identifies the EW row that the particular thread is responsible for summing. Special cases are needed for EW rows with indices greater than the block width. At least one special case is always needed because ew_d is odd. It is expected that choosing an EW with

dimensions closest to a multiple of 16 will limit warp divergence and thus execute faster. The power-of-two constraint on the block width is imposed to allow the use of an efficient warp-based reduction for the final sum calculation. While the solution will accommodate any sized EW, reducing the number of special cases offers faster performance. Therefore the EW sizes for this research expected to provide the fastest performance are chosen such that ew_d is 1 plus a value that is a multiple of 16 that is also a power of 2. EW sizes that do not meet these criteria are chosen for comparison. The block height is based on the chosen block width as $blockHeight = numThreads/blockWidth$ where $numThreads \leq maxThreads$.

A description of the algorithm used for summing the band values in the local EW of all pixels in the data scene is given below in listing Pseudocode 2.

```
1   Host:
2   // Setup GPU and call CUDA Kernel
3       CUDA Initialization
4       Copy data to device and bind to Texture Memory
5       Set Grid and Block dimensions
6       Call Device Kernel to Calculate EW Sums in Parallel
7       Device: Each Block Executes The Following in Parallel
8       // Calculate EW Band Sum (∑ b_i)
9           For each pixel in threadIdx.y subsection of blockIdx.y data row of
            interest
10              For each col in threadIdx.x EW row of interest of blockIdx.x
                band of interest
11                  Sum into Shared Memory EW sum array threadIdx.x
                    location
12              Handle special case row(s) and synchronize threads
13              Apply warp reduce to EW sum array
14              Single thread stores result in device scratch pad at pixel and
                band of interest location
15              Synchronize threads
16  Return control to host
```

Pseudocode 2. Algorithm for GPU Parallelization of EW Sum.

The memory usage and access patterns are an important consideration when applying an HPC problem to a GPU solution. The RX HSI processing algorithm, when applied to a typical HSI data scene, utilizes significantly more memory and more frequent trips to memory than a typical graphics problem for which the GPU is designed. Table 7 shows how the available device memory is utilized for the above algorithm to calculate the EW band sums in parallel on the Quadro FX 4800. In the table *h* and *w* are the kernel block height and width, respectively.

Table 7

Memory Usage for EW Sum Kernel

Variable Name	Data Type	Size (Bytes)	Memory Type	Memory Size
resIdx	uint	4	Local / register	Depends on register availability; 16K 4-byte registers (shared) and 16KB (dedicated) Global Memory Total Local / register bytes used: 40
ewStartRow	ushort	2		
ewStartCol	ushort	2		
pixX	int	4		
pixy	int	4		
pixZ	int	4		
myRow	ushort	2		
firstColVal	short	2		
mySum	int	4		
col	int	4		
row	int	4		

Table 7 (continued).

Variable Name	Data Type	Size (Bytes)	Memory Type	Memory Size
ewSum	int	$h \times w \times 4$	Shared	16KB Total Shared Memory used: $8hw$
lastEwRowSum	int	$h \times w \times 4$		

Local Covariance Matrix Calculation

Upon receiving control back from the device, the host reconfigures the grid layout such that the grid width and height are equal to the data width and height. This configuration is designed to allow each block to compute the RX result for a single pixel. The host then invokes the kernel that will use the EW band sums stored on the device to calculate the covariance matrix and perform the necessary computations to complete the RX algorithm.

Table 8 reflects the chosen grid and block layout for calculating the covariance matrix.

The covariance matrix is a $B \times B$ sized symmetrical matrix where B is the number of spectral bands. This solution recommends the limitation that B is divisible by the size of a half warp. Currently all CUDA compute capabilities share a common warp size of 32. Thus for this solution to be the most effective the data is expected to have the number of bands be a multiple of 16. This recommendation minimizes warp divergence.

Table 8

Grid Layout for Calculating Covariance Matrix with Calculated EW Band Sum

Logical Component	Dimension Size	Index Relationship
Grid Width	Num Data Columns ($dataWidth$)	Data Column – one grid column per data column
Grid Height	Num Data Rows ($dataHeight$)	Data Row – one grid row per data row
Block Width	Largest multiple of $numBands$ such that $maxThreads \geq blockWidth \times blockWidth$	Data Band – one thread per $numBands/blockWidth$ bands
Block Height	Same as Block Width	Same as Block Width

The block layout for this component of the RX algorithm is designed so that each $threadIdx.x$ and $threadIdx.y$ serves as index values for the covariance matrix. Block width and block height are configured to be equal and chosen such that $maxThreads \geq blockWidth \times blockHeight$, where $blockWidth$ is the largest multiple of the total number of bands. When $blockWidth = B$ then i and j in equation Eq. 16 can be thought of as $threadIdx.x$ and $threadIdx.y$, respectively. When $blockWidth < B$, then each $threadIdx.x$ and $threadIdx.y$ serve $B/blockWidth$ bands.

```
1   Host:
2     Set Grid and Block dimensions
3     Call Device Kernel to Calculate Covariance Matrix and Finish RX
      Calculation
4     Device: Each Block Executes The Following in Parallel
5     // Calculate EW Band Pair Product Sums (∑ b_i b_j) Each thread in block
      executes this
6       For all bands b_i where i is based on threadIdx.y operating on
        (numBands/blockWidth) bands
7         For all bands b_j where j is based on threadIdx.x operating on
          (numBands/blockWidth) bands
8           For all pixels in the local estimation window fetched from
            Texture Memory
9             Sum band pair product and store in Shared Memory 2D
              covariance matrix; synchronize threads
10    // Complete Covariance Matrix Calculation
11      If tidX < numBands where tidX = threadIdx.x + threadIdx.y ×
        blockWidth (first numBands threads, i.e. first warps or half warp if 16
        is a multiple of numBands)
12        Fetch EW band sum to Shared Memory sum in Global Memory
          "scratch pad" for all bands of the pixel of interest (x, y, z) based
          on (blockIdx.x, blockIdx.y, tidX); synchronize threads
13      For all bands b_i where i is based on threadIdx.y operating on
        (numBands/blockWidth) bands
          For all bands b_j where j is based on threadIdx.x operating on
          (numBands/blockWidth) bands
14          Calculate mean μ_{b_i} from the sum ∑ b_i in Shared Memory sum
            vector, calculate the product ∑ b_j × μ_{b_i}, and subtract this from the
            EW band pair product sums stored in the Shared Memory
            covariance matrix, store result back into covariance matrix;
            synchronize threads
```

Pseudocode 3. Algorithm for GPU Parallelization of the Covariance Matrix Calculation.

Calculating the RX Result

The final RX result is calculated based on the covariance matrix using the Cholesky decomposition. The Cholesky decomposition is not the bottleneck of the RX algorithm and therefore parallelization of this component is beyond the scope of this research. However, the solution for parallelizing the covariance matrix lends itself to a moderately parallelized approach for completing the RX

computation in that the covariance matrix needed is resident in SM Shared Memory which persists only while a block lives. Therefore, the final RX result calculation is parallelized at the block level, but a combination parallelized / serialized solution is given at the thread level. That is, each block calculates the result for a single pixel in a parallel fashion though only a portion of the available threads in the block are used to calculate the final result. A single thread in the block finishes the RX calculations by calling the Cholesky decomposition method, providing the previously calculated covariance matrix and stores the result into Global Memory.

```
1   Device: (still on device kernel started above in Pseudocode 3)
2     // Calculate Result
3       if tidX < numBands where tidX = threadIdx.x + threadIdx.y × blockWidth
4         Fetch tidX based band of pixel of interest to Shared Memory pixel vector from Texture Memory; synchronize threads
5         Calculate tidX based mean vector μ and subtract from pixel vector and store results in Shared Memory demeaned pixel vector; synchronize threads
6         //Do Cholesky Decomposition
7           If threadIdx.x = 0 and threadIdx.y = 0 (Cholesky decomposition parallelized at the block level, not at the thread level)
8             Solve Ax = b for x using Cholesky Decomposition (where A is the covariance matrix and b is the demeaned pixel); synchronize threads
9         Perform Vector Vector Multiply between resulting vector x and demeaned pixel vector using threads with all tidX < numBands and warp reduce
10        Single thread stores resulting scalar value into output image at current pixel location in 1D array result location in Global Memory
11  Return control to host
12  Host:
13    Copy result from device to host memory
14    Write output image to file
```

Pseudocode 4. Final Calculation of the RX Algorithm.

The final result is calculated in the same kernel as the covariance matrix, thus the grid layout is unchanged. Pseudocode for calculating the final RX result is given in listing Pseudocode 4.

The following table describes the device memory utilization for the kernel that calculates the covariance matrix and completes the RX algorithm computations.

Table 9

Memory Usage for CovRX Kernel

Variable Name	Data Type	Size (Bytes)	Memory Type	Memory Size
resIdx	uint	4	Local / register	Depends on register availability; 16K 4-byte registers (shared) and 16KB (dedicated) Global Memory Total Local / registers used: $8B + 30$ Does not include registers required for calculations / intermediate results or loop counter variables
ewStartRow	ushort	2		
ewStartCol	ushort	2		
ewSize	float	4		
bandY	ushort	2		
bandX	ushort	2		
myCol	ushort	2		
numBandsToWork	ushort	2		
tidX	ushort	2		
p	double	$B \times 8$		
mysum	double	8		

Table 9 (continued).

Variable Name	Data Type	Size (Bytes)	Memory Type	Memory Size
ewDataRow	short	$\sqrt{N} \times B \times 2$	Shared	16KB Total Shared Memory Used: $10B + 2B\sqrt{N} + 8B^2$ Effectively maxes out with 40 bands and 33×33 EW
covMat	double	$B \times B \times 8$		
pixVec	short	$B \times 2$		
difVec	float	$B \times 4$		
xVec	float	$B \times 4$		

Tradeoffs and Assumptions

This research focuses on developing a solution for leveraging the GPU to calculate large numbers of relatively small covariance matrices in parallel. The special case of edge pixels is not addressed in this research. Likewise, this design assumes the input data width and height are greater than the number of spectral bands and that the image is square. Although the CUDA API supports dynamic grid and block size allocation, these values are defined for compile time optimization.

In an attempt to tailor the RX algorithm for the GPU, a constraint on the number of bands is imposed. This constraint requires that the number of spectral bands (B) included in the input data is divisible by 16 and be small enough to allow a full $B \times B$ covariance matrix to fit into SM Shared Memory. The former limitation on the number of bands facilitates a solution that minimizes warp divergence and thus maximizes concurrent thread execution for covariance matrix calculations. The latter band number limitation reduces the memory

access latency required by the covariance matrix calculations, thereby improving the speed performance of the calculation.

A tradeoff that reduces the speed performance for calculating the covariance matrix is the decision to perform double precision operations for this component. Double precision calculations require more time than single precision operations in any CUDA compute capability, but the precision is needed to calculate an accurate covariance matrix and ultimately the RX result. Likewise, storing the covariance matrix as 64bit doubles utilizes twice as much Shared Memory as floating point values would use, which in turn reduces the available Shared Memory that could be used for an on-chip EW solution, for a small EW.

The decision to create a solution that keeps the EW in global Texture Memory rather than on-chip for fast Shared Memory access allows the solution to operate with EW sizes that are typical of HSI data processing and too large to fit into compute capability 1.x 16KB Shared Memory. For some applications that do not require a larger EW and leveraging the 48KB of Shared Memory available on the Fermi with compute capability 2.0, this restriction could be lifted and a faster solution using on-chip EW storage could be implemented.

Experimental Setup

Sample Data Set

A hyperspectral scene collected in Mobile, AL by the Army Corps of Engineers with a CASI 36 band hyperspectral sensor is used as the sample input data set. This data is subsampled in all dimensions for the purposes of this research. While target detection performance is the ultimate goal of HSI

processing and while subsampling in the spectral domain may negatively affect the detection performance of the RX algorithm implemented, the primary focus of this research is working toward improving the speed of spectral covariance matrix calculations. To that end, the data set is sized such that it maps easily to the Quadro FX 4800 compute capability. A subsection of the data set that is not specifically designed to map to the Quadro is also chosen for comparison. A true-color composite of the scene is shown in Figure 1. Execution with the serial implementation of the RX algorithm using the chosen subsampled data sets provides reasonable and expected detection results. Detailed analysis for improved detection results is beyond the scope of this research.

Table 10 shows the chosen parameter values for timing comparisons.

Table 10

Parameter Values for Empirical Study

Data Width & Height	Number of Bands	Estimation Window Dimensions	EW Sum Kernel Block Width & Height	CovRX Kernel Block Width & Height
512 x 512	36, 32, 30, 16, 15, and 8	33 x 33, 21 x 21, 17 x 17, and 9 x 9	32 x 16, 32 x 8, 20 x 25, 20 x 20, 16 x 32, 16 x 16, 16 x 8, 8 x 64, and 8 x 32	18 x 18, 9 x 9, 16 x 16, 8 x 8, 15 x 15, 10 x 10, 5 x 5, and 4 x 4
256 x 256	36	33 x 33, 21 x 21, 17 x 17, and 9 x 9	32 x 16, 32 x 8, 20 x 25, 20 x 20, 16 x 32, 16 x 16, 16 x 8, 8 x 64, 8 x 32, and 8 x 8	18 x 18 and 9 x 9
128 x 128	36	33 x 33, 21 x 21, 17 x 17, and 9 x 9	32 x 16, 32 x 8, 20 x 25, 20 x 20, 16 x 32, 16 x 16, 16 x 8, 8 x 64, 8 x 32, and 8 x 8	18 x 18 and 9 x 9

Host System Specifications

Table 11 gives the specifications for the computer used in this research.

Table 11

System Specifications

Component	Specification
Operating System	Ubuntu 10.04 Linux
Desktop Environment	GNOME 2.30.0
Kernel Version	2.6.31-20-generic
GCC	4.4.3 (x86_64-linux-gnu)
CPU	Intel® Core™ i7 920 @ 2.67GHz
Number of CPUs	8
CPU Clock	1600MHz with 8
CPU Cache	8192KB
RAM	6.0GB at 2.67GHz
Addressing Width	64 bits
Graphics Card	NVIDIA Quadro FX 4800
Video Bus	PCI-E 16x
Video RAM	1536MB
GPU Frequency	300MHz
Video Driver	NVIDIA Unix x86_64 Kernel Module 256.35

CHAPTER IV

RESULTS

The following discussion presents the timing results and associated analysis for the GPU implementation of the HSI RX anomaly detection algorithm; first in comparison with the baseline serial implementation followed by a more detailed exploration of parameters that affect the speed performance of the parallelized implementation.

Recall, the serial code is a non-optimized brute force implementation of the algorithm for relative comparison with the non-optimized brute force implementation on the GPU. Actual speed performance of the RX algorithm on similar data sets using a more optimized solution is notably faster.

Serial and GPU timing comparisons are based on total processing time. Additional code segments of the GPU implementation timed and analyzed include: CUDA initialization time, time for the EW Sum kernel to execute, time for the kernel that completes the covariance calculations and computes the RX result to execute (CovRX), data transfer from host to device time and time to transfer result image from device to host. Independent variables affecting timing results include the number of bands, size of the EW, number of threads, and the thread block layout. Unless otherwise noted, the input data size is 512×512 for the width and height dimensions.

Serial Results and Comparison Analysis

To form a basis for timing comparisons a serial implementation is created and executed for data sets with varying numbers of bands configured for a

33×33 estimation window (EW). Processing times for data sets with 8, 16, 30, 32, 33 and 36 bands are determined. Additionally, execution of the serial implementation with the input data set consisting of 16 bands is performed varying the size of the EW. Speed performance is analyzed for EWs with the following sizes: 9×9, 16×16, 21×21, and 33×33.

Serial Implementation Results

Figure 14 reports the total processing time the serial implementation requires based on the number of bands in the input data set. The regression line displayed offers an R^2 value of 0.9005, indicating a fairly stable linear trend to require more processing time as the number of bands in the data set increase.

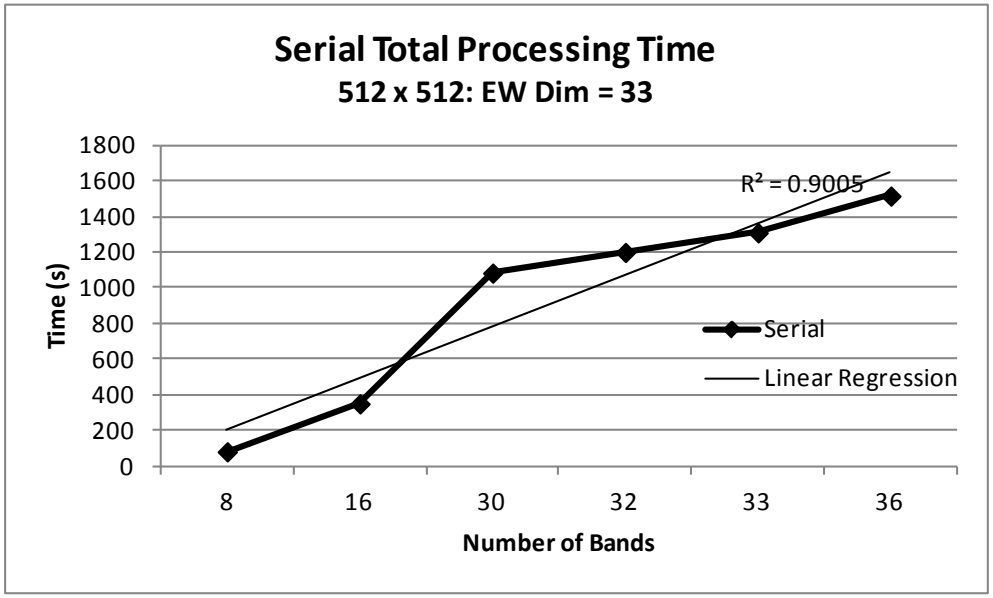

Figure 14. Total Serial Processing Times for the RX Algorithm.

Figure 15 shows that total processing time tends to increase as the EW size increases. This tendency is seen regardless of the number of bands processed. The graph indicates results for the 16 band 512×512 data set.

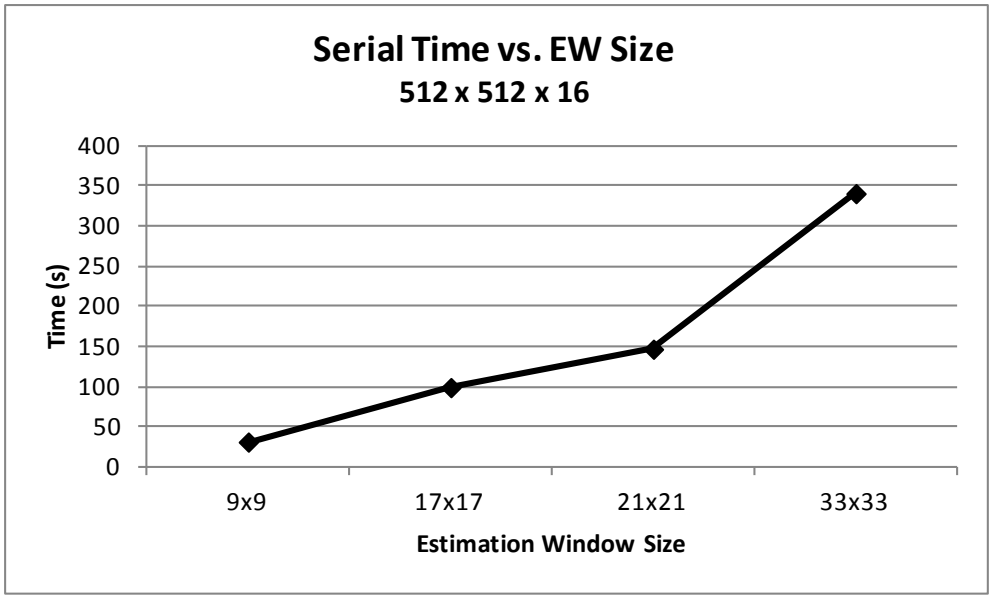

Figure 15. Total Serial Processing Time Comparison as EW Size Changes.

GPU Implementation Results

Figure 16 depicts the total processing times the naïve GPU implementation requires for processing the 512×512 data set with varying numbers of bands. The R^2 value shown with the regression line reveals that while processing times generally increase as the numbers of bands increase this pattern is not as predictable as it is with the serial implementation. The graph reflects results for an EW size of 33×33.

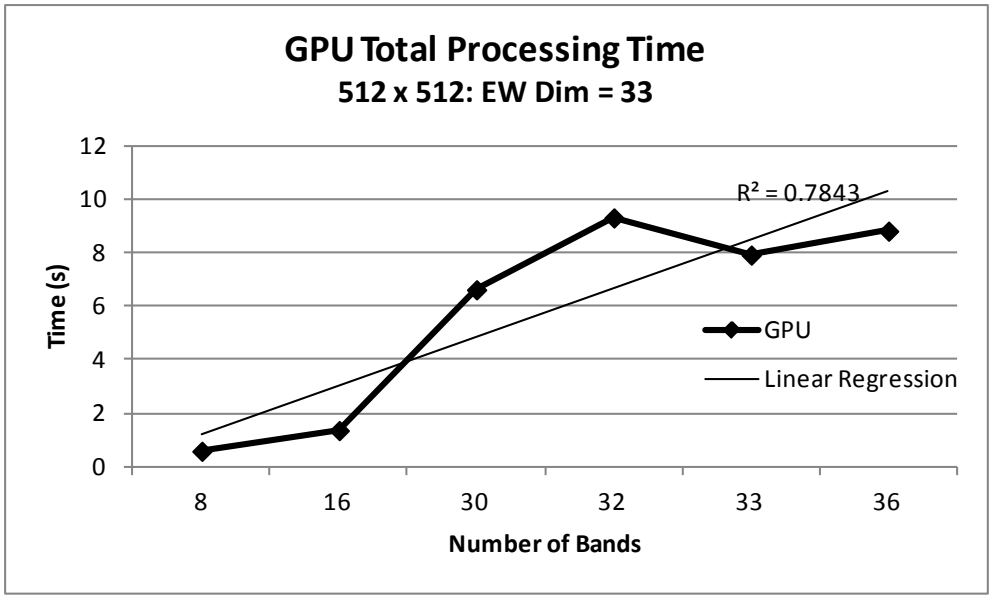

Figure 16. Total GPU Processing Times for the RX Algorithm.

As with the serial implementation, when the EW size increases the total processing time on the GPU also increases, see Figure 17. The graph indicates results for the 16 band 512×512 data set.

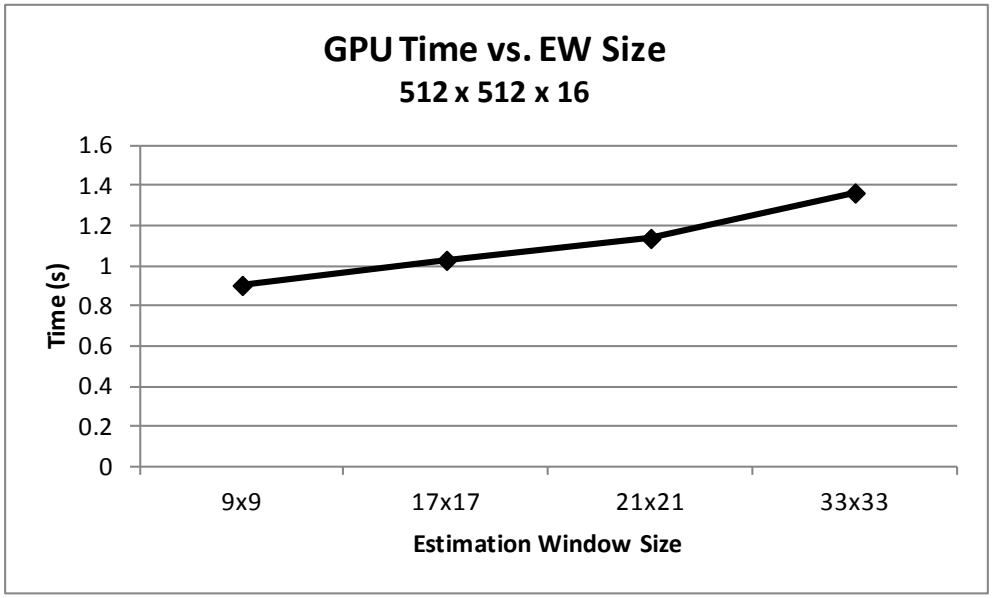

Figure 17. Total GPU Processing Time Comparison Based on EW Sizes.

Serial / GPU Comparison

A speedup metric is not calculated because the serial implementation is not optimized. Nevertheless, the ratio of the timing results of the serial implementation with the timing results of the GPU version provides insight to the speed performance improvements that can be realized with a GPU implementation.

Figure 18 illustrates that the speed performance improvement realized by the GPU implementation is relatively consistent regardless of the number of bands. However, the jump in relative improvement when the data set has 16 bands indicates that the GPU implementation is slightly more optimized for 16 band data than for other numbers of bands. Considering that data transfer within the chip is designed for the half-warp, which is 16 threads, this performance improvement is not surprising. It is however, surprising to see the performance improvement for the 32 band data set is not as high as for other tested data sets considering that a warp is 32 threads and the hardware is optimized to work with multiples of 32. Nevertheless, this code has not been optimized and thus an optimized solution may capitalize on the warp size for improved performance.

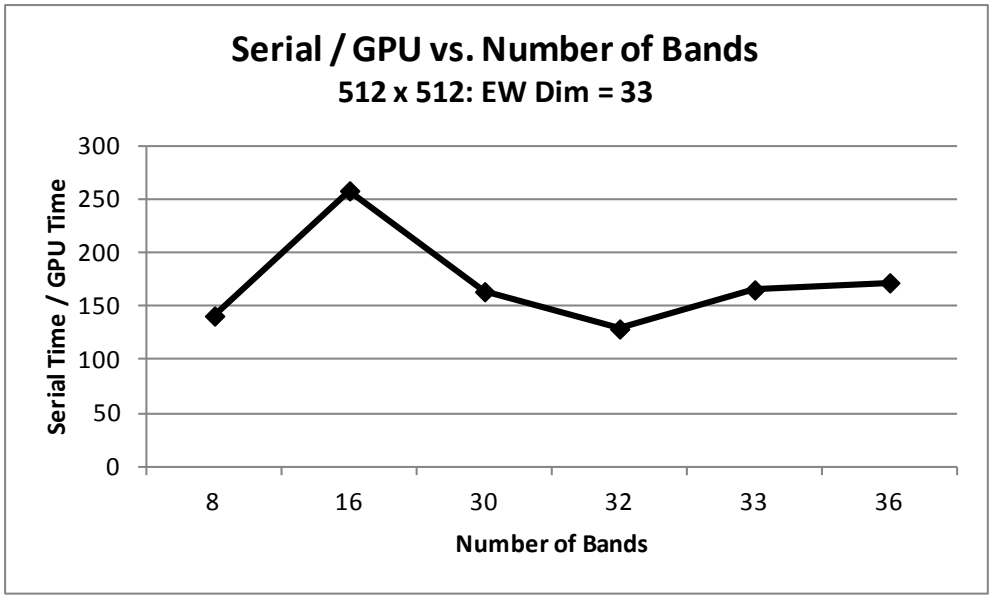

Figure 18. Serial / GPU Processing Time Ratio for Varying Numbers of Bands.

Unlike the consistent speed advantage across increasing numbers of bands, the speed advantage of the GPU implementation as compared to the serial implementation increases when the size of the EW increases. Figure 19 below shows this trend for the 16 band data set.

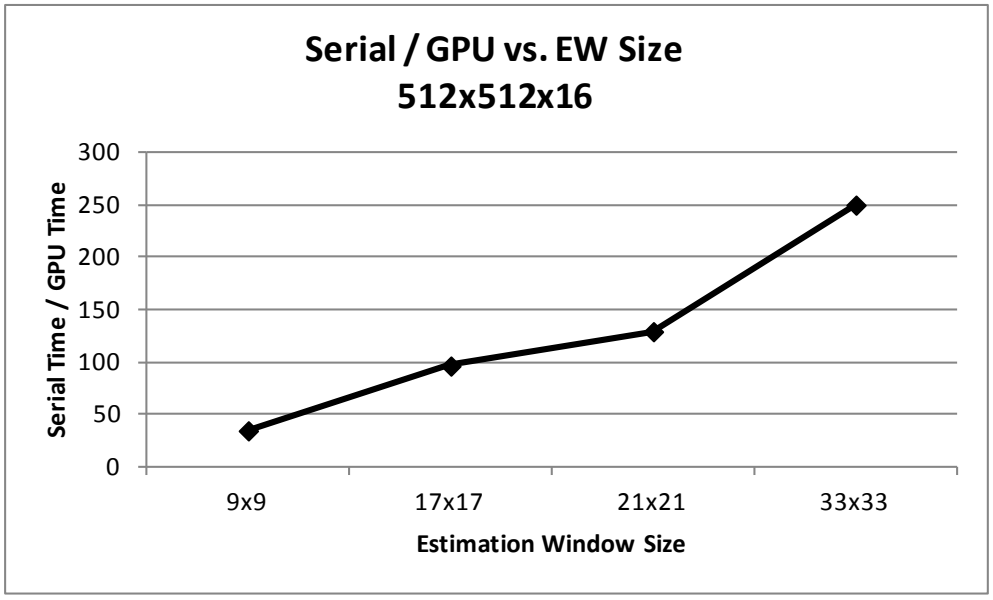

Figure 19. Serial / GPU Processing Time Ratio for Varying EW Sizes.

GPU Results and Analysis

Figure 20 graphs the total execution time on the GPU as the EW size changes for data sets with different numbers of bands. This illustrates the significance the EW size has on speed performance. For example, the total processing time for the 36 band data set with a 9×9 EW is almost half the total processing time for the 30 band data set with a 65×65 EW. While when the EWs are the same the 30 band data set is processed a third again as fast as the 36 band data set.

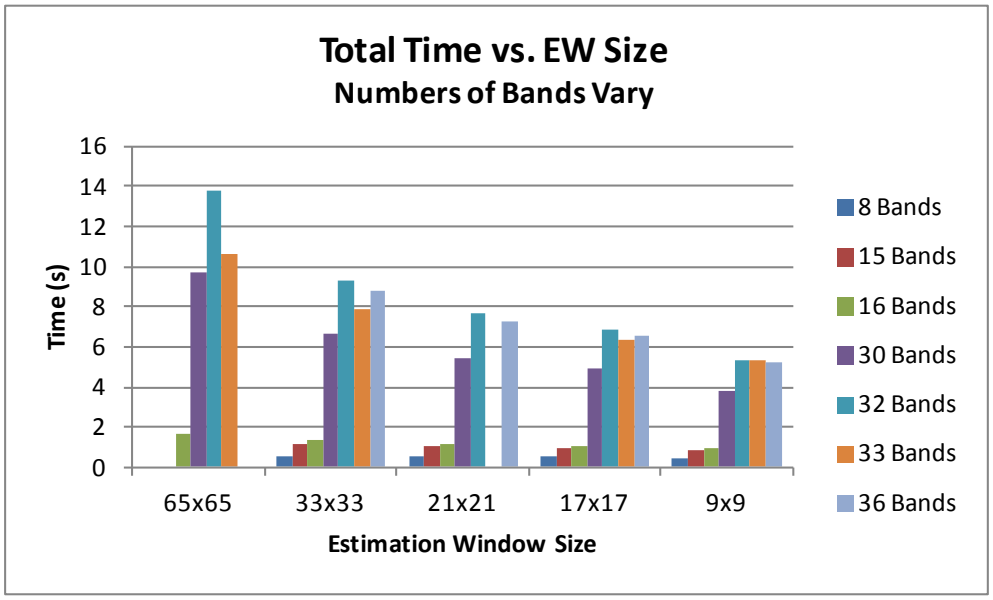

Figure 20. Total GPU Processing Time.

The device initialization and CovRX components of the GPU implementation require the most time of the components timed. These are discussed in more detail below, however Figure 21 charts the total time less the CUDA initialization time against a representation of the GPU kernel block layouts for different numbers of bands. The timing values included in this chart are results of processing a 512×512 data set using a 17×17 EW. The data point

labels displayed represent the kernel block layout. The first value indicates the EW Sum kernel block height. The EW Sum kernel block width for all data points is 16. The second value in the displayed pair is the CovRX block width and indicates the size of the CovRX kernel block. The CovRX kernel block threads are configured into a square so that the block height equals the block width.

The zigzag nature of the plotted lines in Figure 21 reveal the significance the block layout has with regard to speed performance of the CovRX kernel. Notice that the CovRX block width value dictates the charted speed performance. As the block width decreases the total time increases. Conversely, as the time decreases the block width increases. The EW Sum block layout similarly has an effect on speed performance; however the CovRX kernel time dominates at this level of analysis. A detailed exploration of the EW Sum block layout is given below.

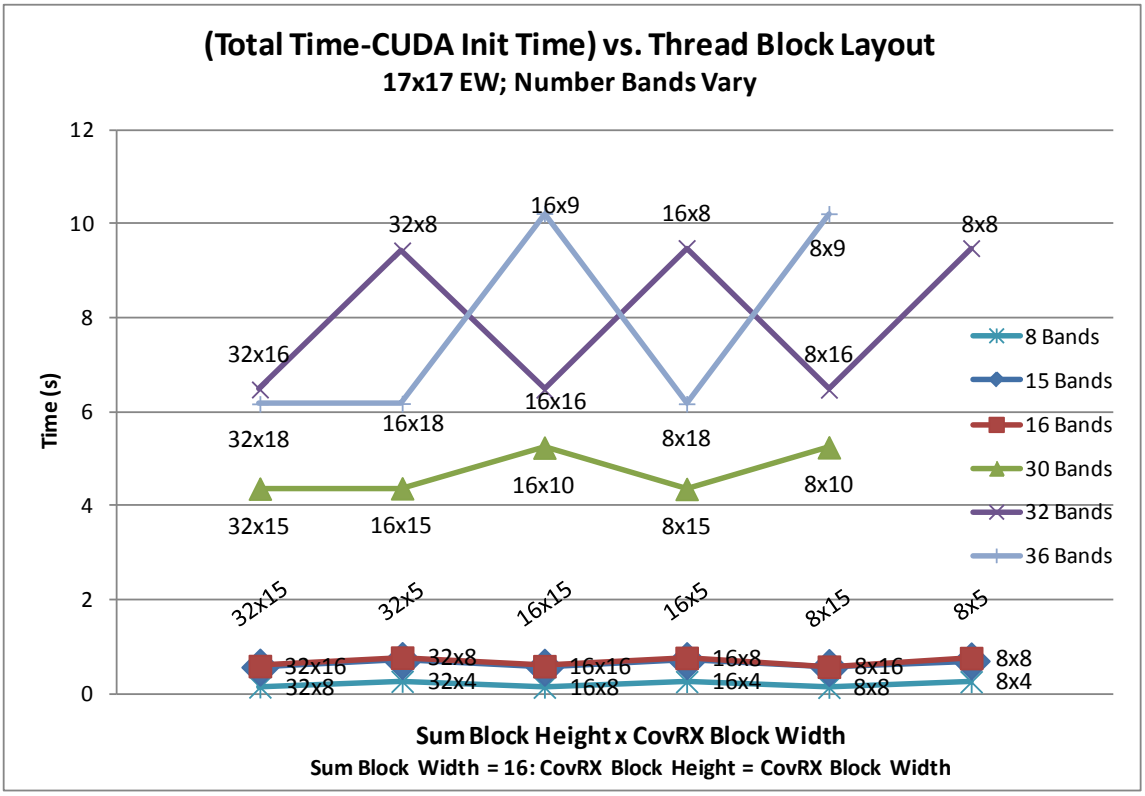

Figure 21. GPU Processing Time for Different Kernel Block Layouts.

CUDA Initialization

Setting the GPU up for code execution involves a significant amount of overhead, most noticeably the CUDA initialization time. While this has a considerable effect on the total time performance, for stream processing it becomes inconsequential since initialization is only necessary once. All successive images processed do not bear this penalty. In general the time required for CUDA initialization is very similar regardless of the number of bands in the input data or the size of the EW, both of which affect the overall speed performance. Figure 22 shows the CUDA initialization time for the various numbers of bands and EW sizes tested.

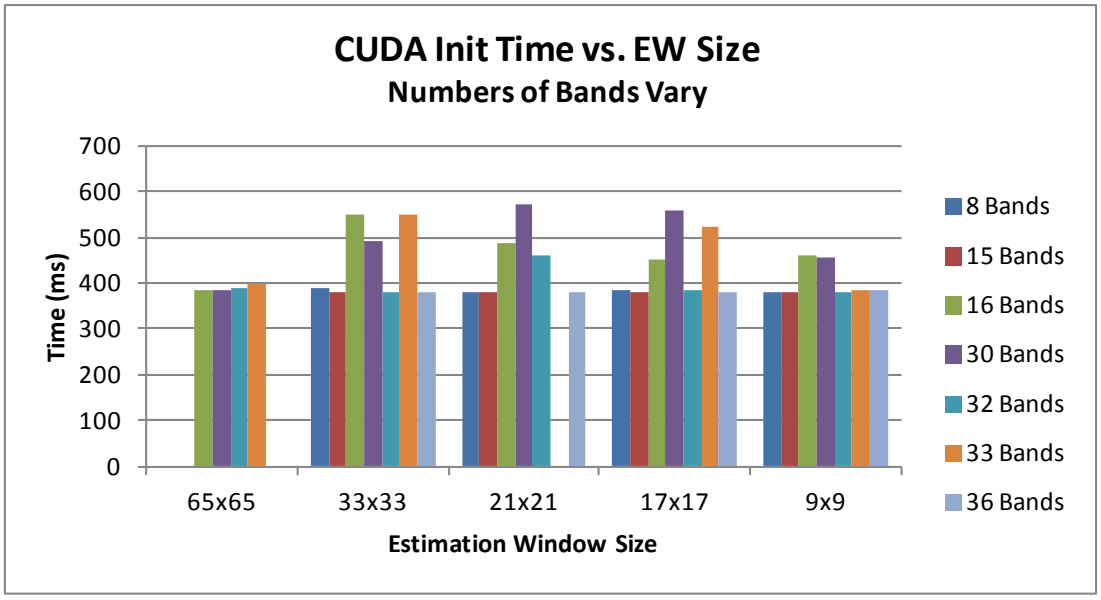

Figure 22. CUDA Initialization Times.

Similarly, varying the size of the input data with respect to the width and height, for powers of two, minimally affects the CUDA initialization time, as seen in Figure 23.

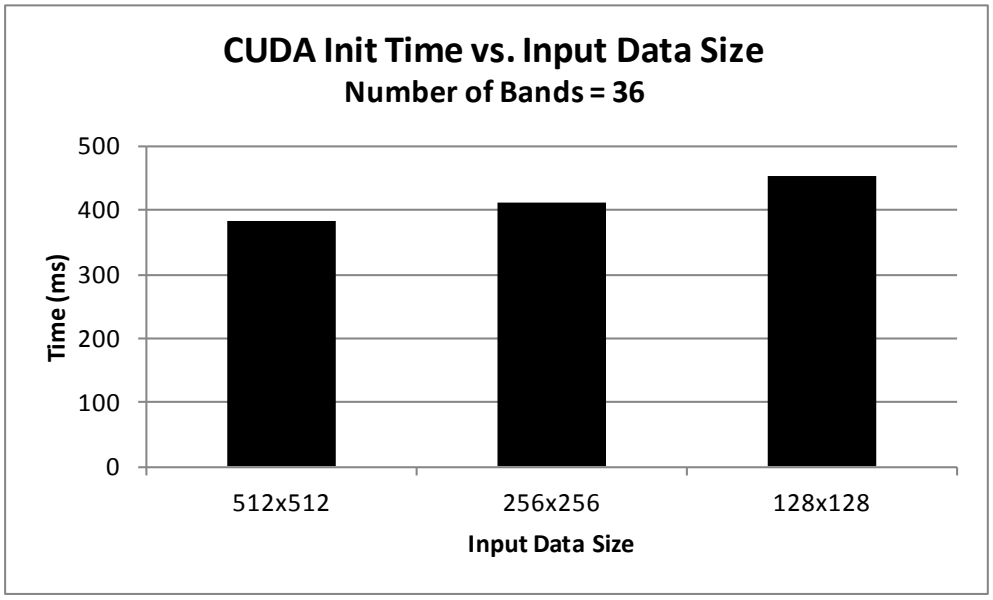

Figure 23. CUDA Initialization Time for Different Sizes of Input Data.

While the block layout and total number of threads in a block may affect the total speed performance, Figure 24 and Figure 25 make it clear that these

variables, with respect to the EW Sum kernel and CovRX kernel, have little effect on the initialization time. Figure 24 shows the CUDA initialization time with different EW Sum block layout configurations that result in different numbers of threads for the EW Sum kernel. Similarly, Figure 25 shows the CUDA initialization time with different CovRX block sizes. Both figures reflect graphs for 36 bands (a), 8 bands (b), and 15 bands (c). Axis descriptions and legends are the same for all graphs displayed for each figure.

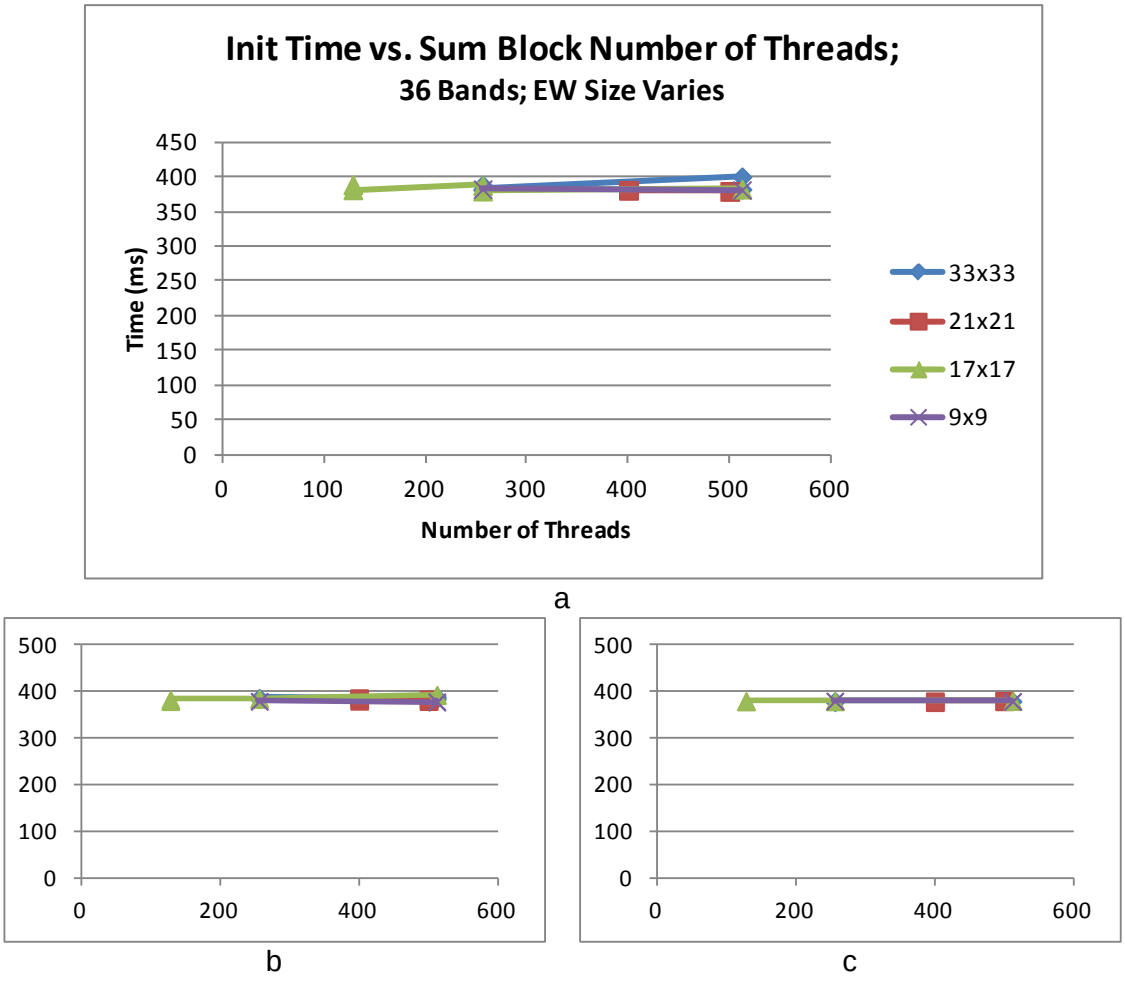

Figure 24. CUDA Initialization Times as EW Sum Block Numbers of Threads Vary.

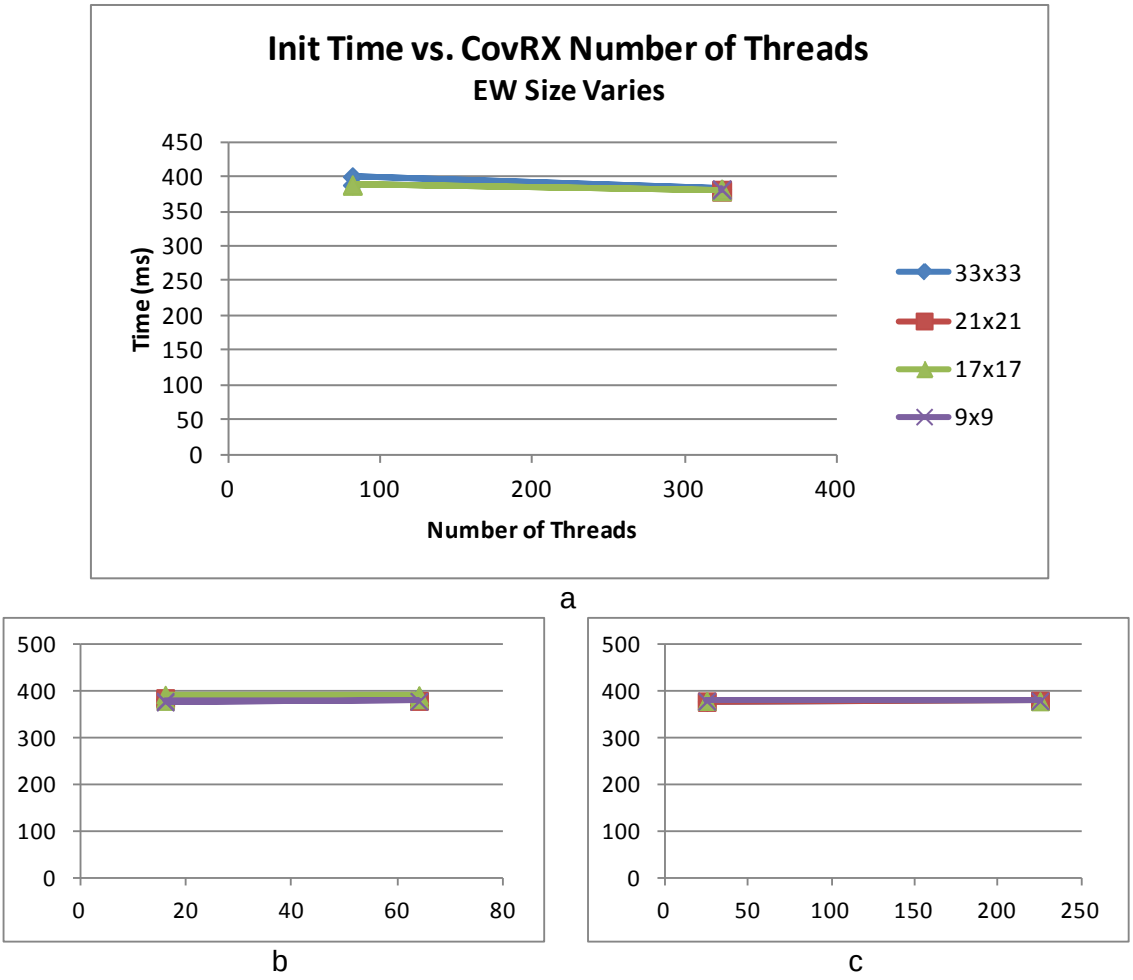

Figure 25. CUDA Initialization Times as CovRX Block Numbers of Threads Vary.

Estimation Window Sum Kernel

The EW Sum timing results exhibit the expected behavior in that the time increases as the number of bands to process increase. Figure 26 illustrates this behavior with the 512×512 data set and varying numbers of bands for a 17×17 sized EW.

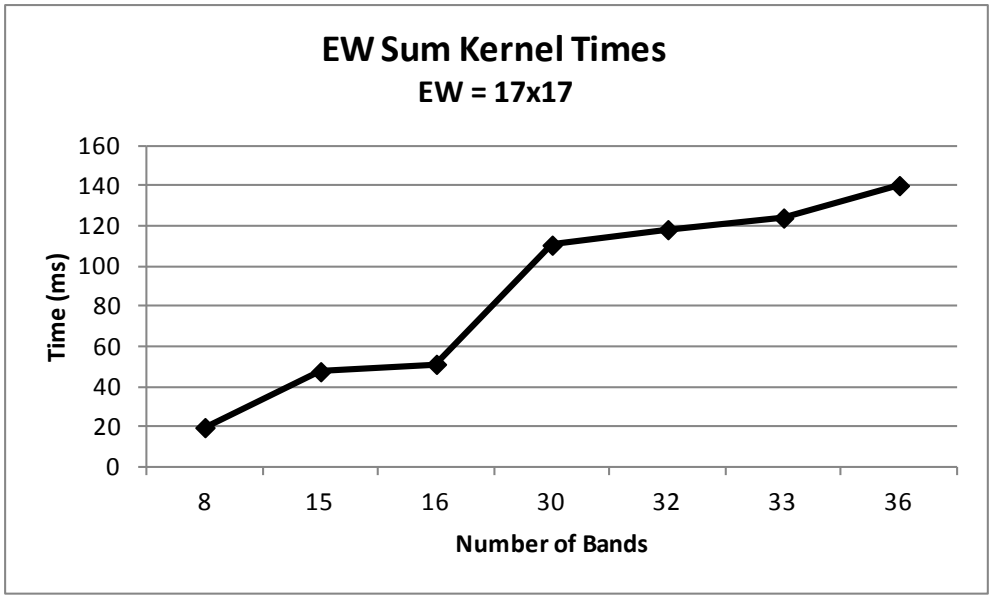

Figure 26. EW Sum Kernel Times.

As mentioned earlier, the block layout for the EW Sum kernel affects the speed performance for summing the band values of all pixels in the EW. Recall the design of this kernel allocates threads in the x dimension to operate on the x EW row while the threads allocated in the y dimension define which subsection of the data row the y block row processes. This configuration requires the threads on each block row to sum the EW of more than one pixel and thus the summing task is not fully parallel, i.e. all pixels the block is responsible for processing are not processed at the same time.

Figure 27 shows that as the number of threads allocated to the EW Sum block increase, the processing time can increase. This is especially noticeable as the EW size increases, note the 33×33 EW results. As the EW size decreases this apparent performance degradation is not consistently exhibited, shown below by the 9×9 sized EW. The figure shows the processing time for the EW Sum kernel with different numbers of threads, different EW sizes and 36 bands

(a), 8 bands (b), and 15 bands (c). Axis description and legends are the same for all graphs displayed.

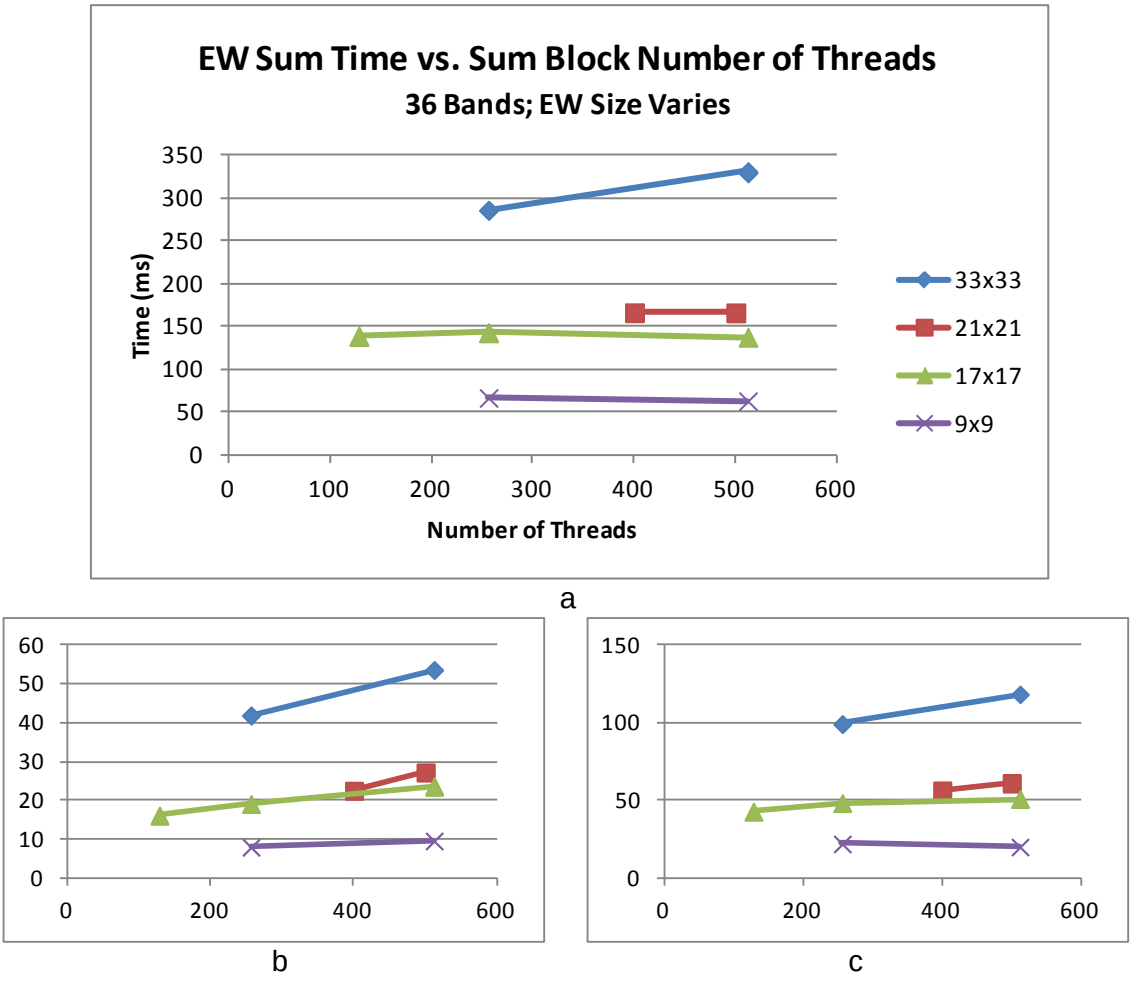

Figure 27. EW Sum Kernel Processing Time with Different Numbers of Threads.

While comparing the EW Sum speed performance with respect to the number of threads, it is interesting to note that as the difference between the block width and the block height increases, the speed performance increases. This is shown in Figure 28 which plots the ratio of the block width to the block height; as the ratio increases the processing time decreases. The figure depicts processing times for the EW Sum kernel with different block width : height ratios,

different sized EWs and 36 bands (a), 8 bands (b), and 15 bands (c). Axis description legends are the same for all graphs displayed.

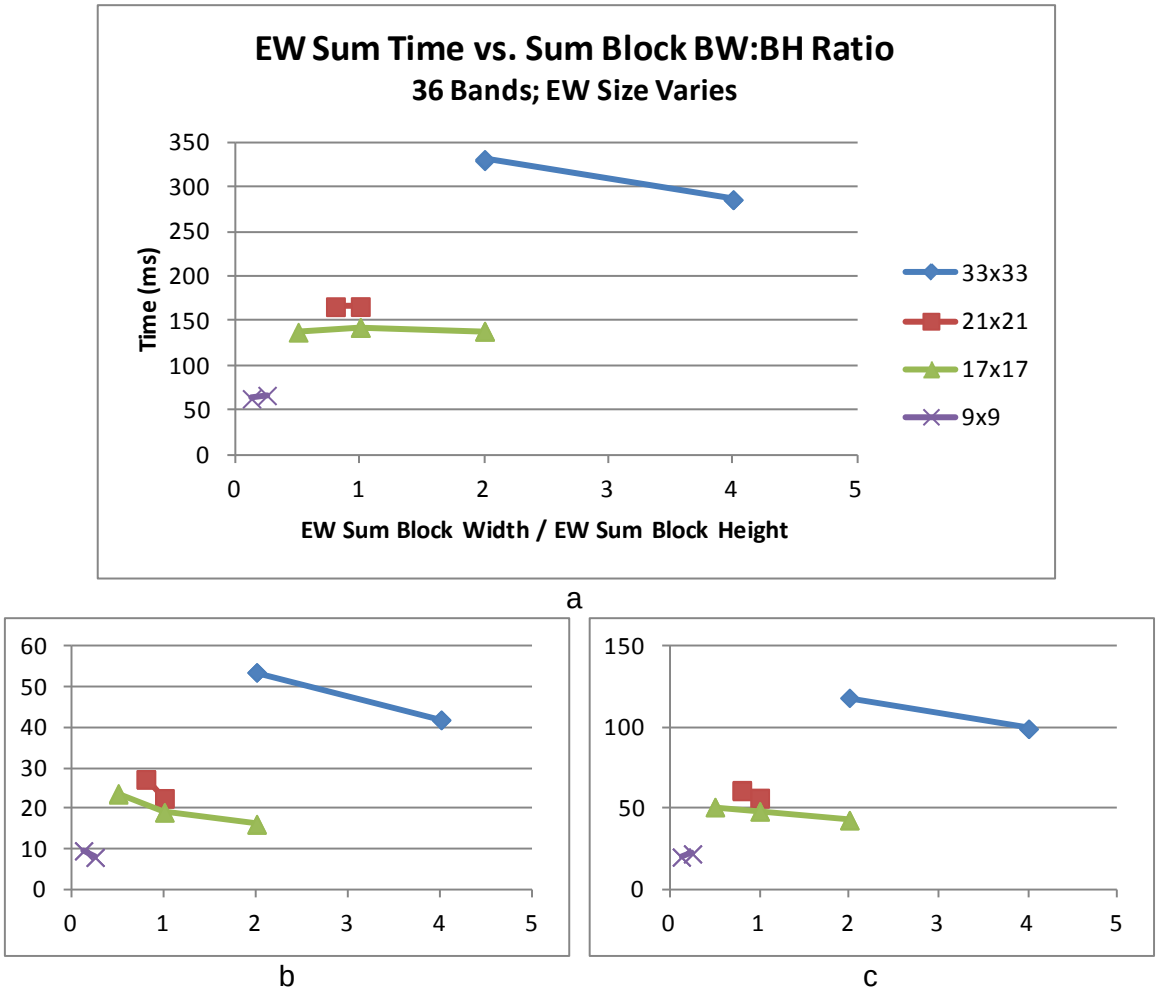

Figure 28. EW Sum Kernel Processing Times with Different Block Width : Height Ratios.

Figure 29 illustrates the exponential behavior of the time required to sum the pixels in the EW as the EW size increases. This exponential increase in processing time is consistent regardless of the number of bands in the input data set.

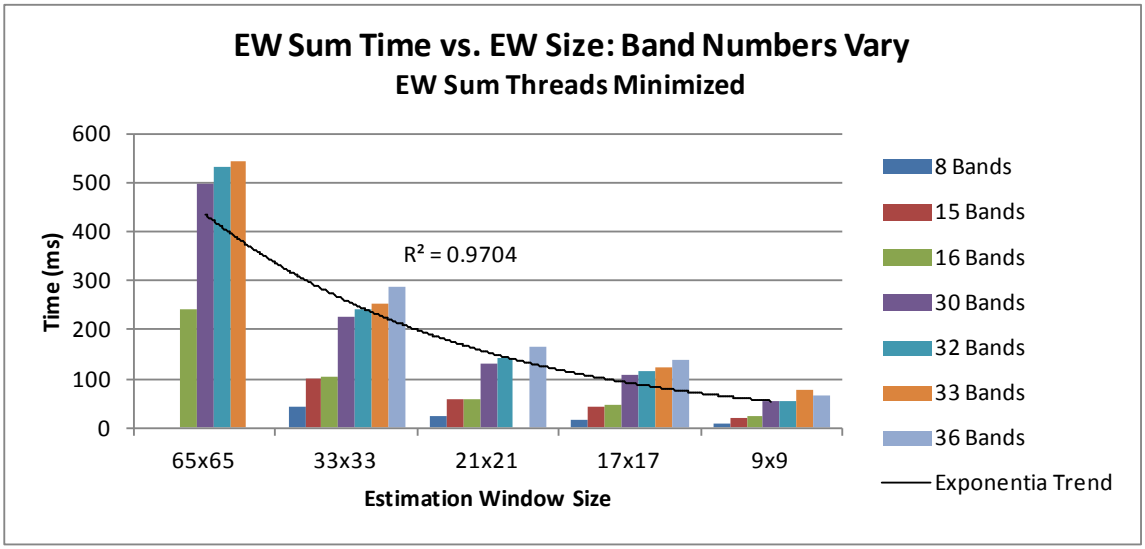

Figure 29. EW Sum Kernel Times for Different Size EWs and Numbers of Bands.

Covariance Completion and RX Calculation Kernel

Like the EW Sum kernel timing results, the time requirements for the CovRX kernel increase as the number of bands increase, with an unexpected spike when the input data has 32 bands. This response is shown below in Figure 30.

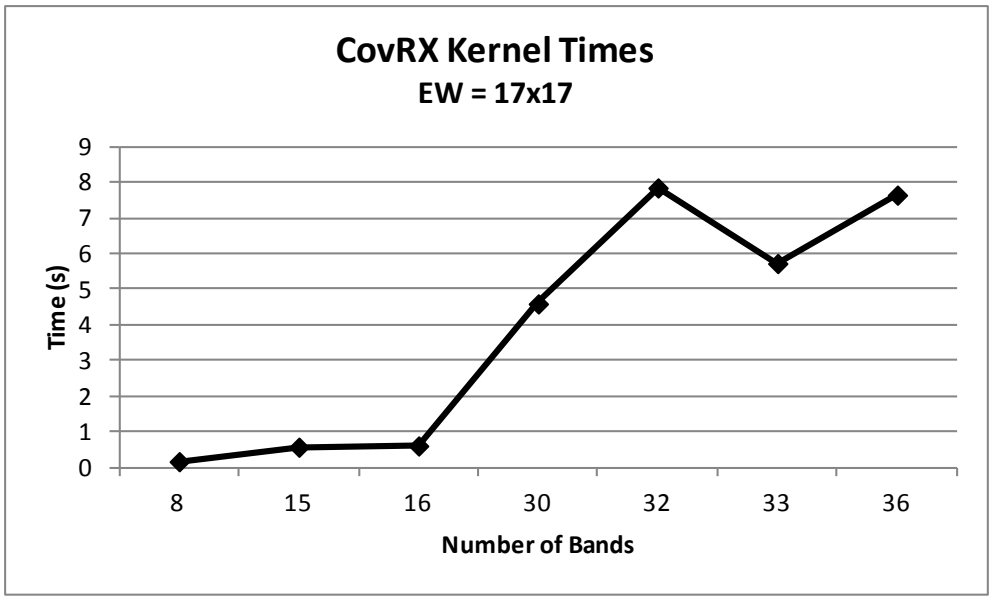

Figure 30. CovRX Kernel Times for Different Numbers of Bands.

The CovRX kernel assigns a single pixel to a single block to complete the calculation of the covariance matrix using the EW Sum values calculated and stored in global memory by the EW Sum kernel and then computes the RX result using the covariance matrix stored in the blocks shared memory. The thread layout corresponds with the size of the covariance matrix which is a square $B \times B$ matrix where B is the number of bands. The block width equals the block height and is some multiple of the number of bands in the input data set.

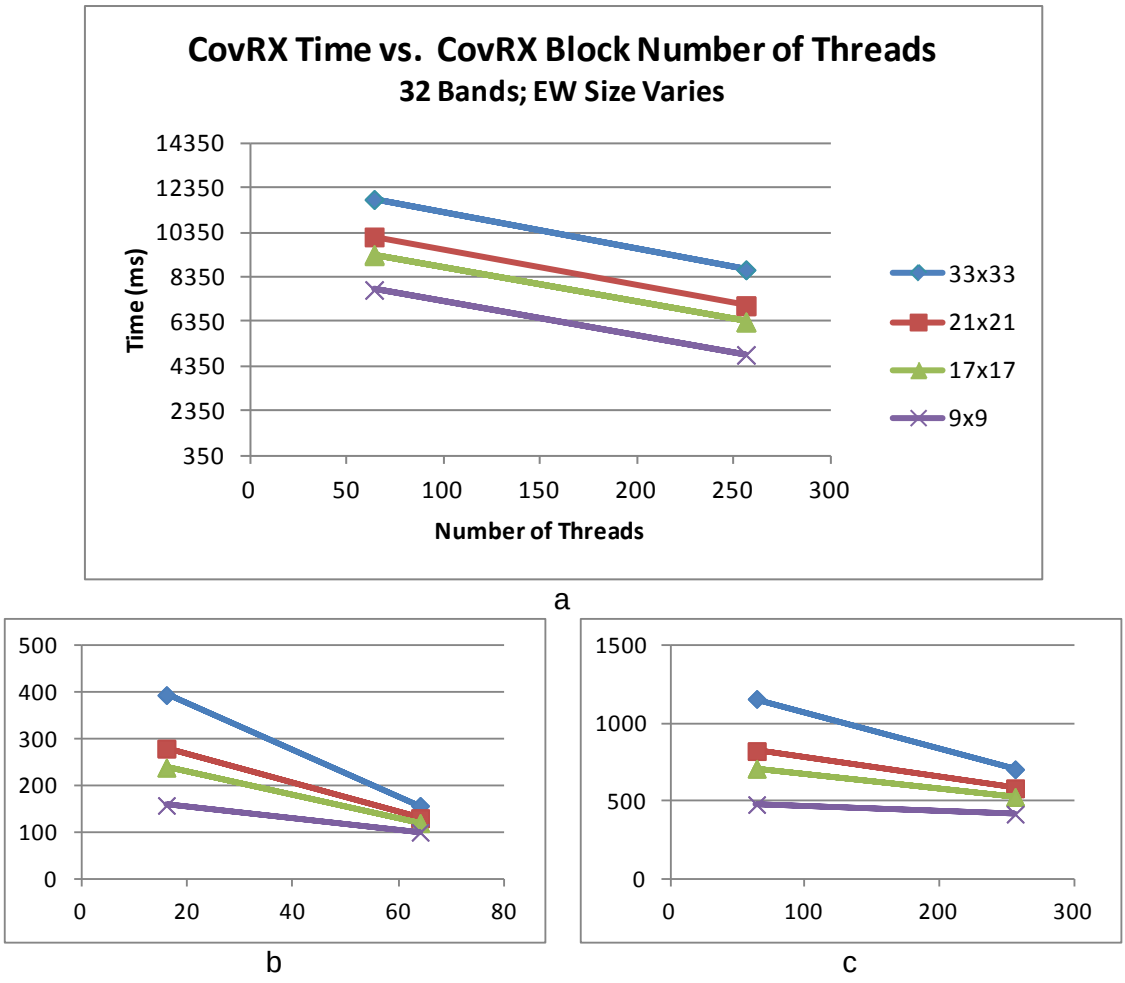

Figure 31. CovRX Kernel Times with Different Numbers of Threads and EW Sizes.

Unlike the EW Sum kernel which requires more processing time when assigned more threads, the CovRX kernel performs faster with more threads. Figure 31 shows the decrease in processing time as the numbers of threads increase for various sized EWs and three data sets with different numbers of bands; 32 bands (a), 8 bands (b), and 16 bands (c). Axis description legends are the same for all graphs displayed.

Processing time for the CovRX kernel increases exponentially as the EW size increases based on the EW sizes tested. Given a specific EW size with different numbers of bands the speed performance is dependent upon the implementation details that may be more optimal for some input data sizes and less optimal for others. Figure 32 below demonstrates this trend.

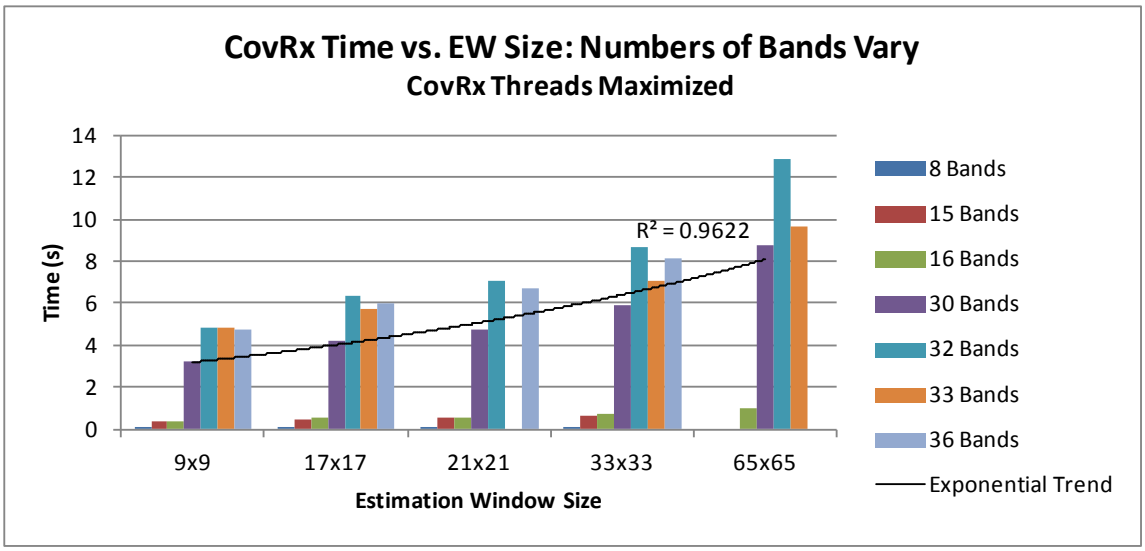

Figure 32. CovRX Kernel Times with Different EW Sizes and Numbers of Bands.

Data Transfer

HSI Data from Host to GPU

The size of the data transfer from the host to the GPU device linearly affects the transfer time. Data sets with more bands require more transfer time,

as expected. Figure 33 shows the time required to transfer the input data from the host to the device for different numbers of bands and different EW size configurations. This chart illustrates the expected behavior that the EW size configuration has little effect (times vary within approximately 1ms) on the data transfer times while increased numbers of bands increases the transfer time required.

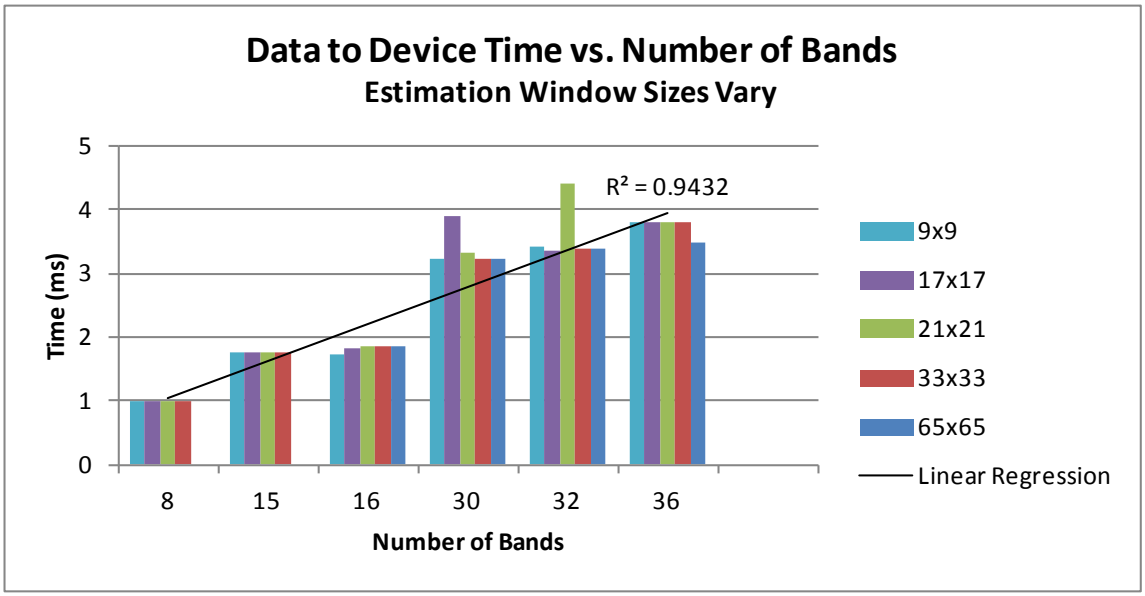

Figure 33. Data Transfer Times From Host to Device.

The input data are 16 bit short integer values. The total number of input data bytes transferred from the device to the host is therefore determined by

$$t = W \times H \times B \times 2 \qquad \text{Eq. 17}$$

where t is the number of bytes transferred, W and $H$are the data width and height, respectively, both 512 in this case, and B is the number of bands. Thus for the 36 band input two-byte data, it takes close to 3.5ms to transfer 18MB.

RX Result Image from GPU to Host

As expected the image width and height affect the transfer time from GPU to host, however the RX result image is the same size regardless of the numbers of bands in the input data and is similarly unaffected by the EW size. Figure 34 reveals this expected behavior for the result image data transfer time from device to host for a 512×512 image. This graph shows times for input data with different numbers of bands and various EW size configurations.

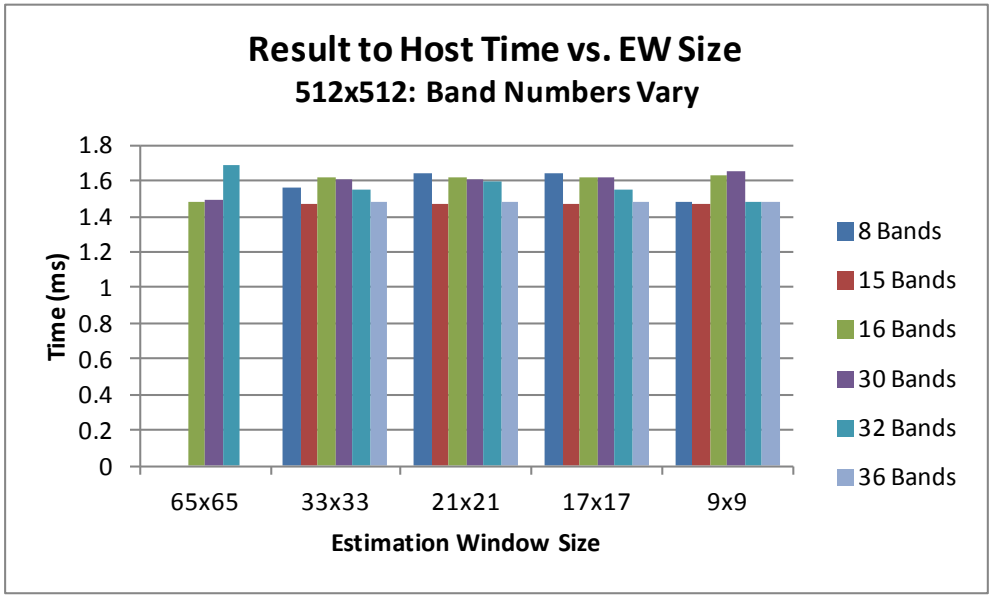

Figure 34. Result Image Data Transfer Times from Device to Host.

The RX results are returned as a floating point image. Accordingly, the total number of bytes transferred from the device to the host is determined by

$$t = W \times H \times 4 \qquad \text{Eq. 18}$$

The result image size for the processed input data represented in Figure 34 above is 1MB and is the same regardless of EW size used for processing and numbers of input bands.

To / From Data Transfer Relationship Analysis

Figure 35 illuminates that for smaller numbers of bands the transfer time from the device back to the host takes longer than the original transfer of input data from the host to the device. This apparent idiosyncrasy is due to the fact that the data bus from the device to the host is much slower than the bus from the host to the device (NVIDIA 2009b). As the numbers of bands increase, however time for transferring data to the device begins to dominate.

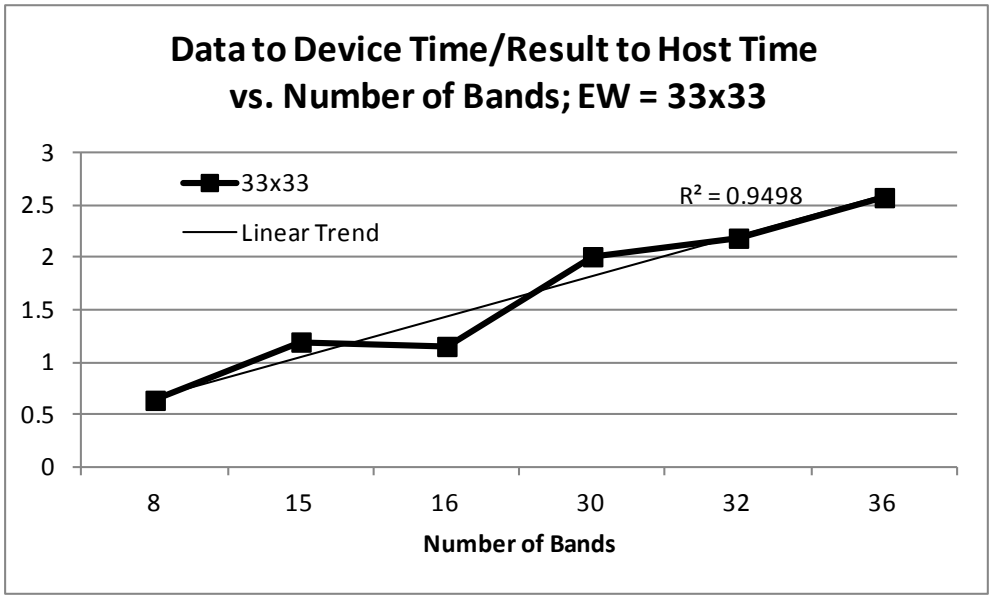

Figure 35. Quotient of the Data-to-Device Time with the Results-to-Host Time.

As with the fewer numbers of bands, input data sets with smaller width and height dimensions require more time for the return trip to the host with the result image than for the transfer of the input data from the host to the device. This is illustrated in Figure 36 by the ratio value below 1 for the 128×128 input data. The figure also shows that at input data sizes of approximately 256×256 the transfer time for the result back to the host begins to require less time than transfer time for the input data from the host to the device. Figure 36 shows the

data transfer times for different sizes of input data from host to device, result data from device to host, and the ratio of these times. The data sizes being transferred are also indicated on the chart. While the data sizes themselves are different, the ratios of input data size with result size are all the same at 18.

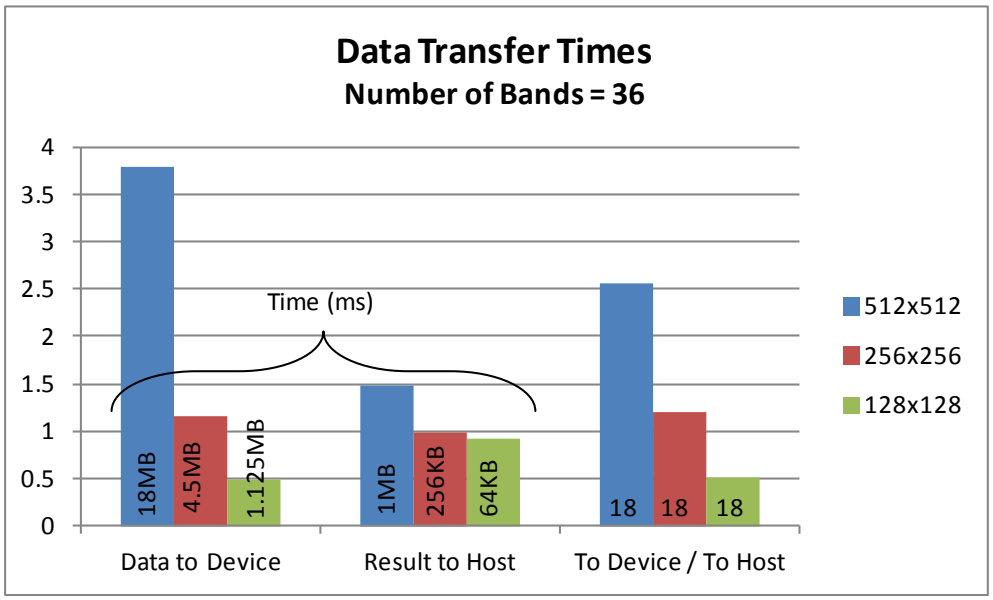

Figure 36. Data Transfer Times and Sizes.

Performance Analysis

It is shown that the total RX algorithm processing time with the GPU implementation is much faster than the processing time with the serial implementation. The more detailed timing results of the different components that make up the GPU implementation verify the original assessment that the covariance matrix calculation is by far the bottleneck of the algorithm and continues to be with the GPU implementation.

The GPU implementation also introduces timing considerations not required with the serial implementation. Namely, the CUDA initialization time and data transfer times to and from the device. Nevertheless, even with these

additional timing considerations the GPU implementation is faster than the serial implementation. Applications requiring multiple consecutive scenes such as in a real-time data collection and processing scenario will only take this time penalty once at the beginning of each session. Also, the initialization time is relatively stable and can therefore be easily accommodated when planning a data collection event. It is expected for the data transfer from the host to the device to take longer than transferring the results back to the host. It is shown above, however that for smaller data sets this may not be the case.

The fundamental designs for the EW Sum kernel and the CovRX kernel differ in that the EW Sum kernel requires a block to handle multiple pixels while the CovRX kernel has a $1:1$ allocation of blocks to pixels. That is, the EW Sum kernel is not parallelized to the same degree as the CovRX kernel. The behavior of these two kernels differ in that the processing time for the EW Sum kernel increases as the number of threads allocated for it increase, while the processing time for the CovRX kernel decreases as the number of threads increase. This implies that a balance between fully parallelized and un-parallelized may offer improved performance for the CovRX kernel.

While the CovRX kernel block layout is square, the EW Sum block layout is such that the block height and block width may be different. The results shown above reveal that faster performance is realized as the block width becomes increasingly bigger than the block height. This is reasonable considering the block height derives the number of pixels processed serially and therefore is affected by the half-warp-size driven memory access heuristics. As the block

height increases, the serialization decreases and caching from coalesced memory accesses are better utilized.

The CovRX kernel by far dominates the required processing time which is anticipated. It is seen in the discussion above that as the number of bands increase the time for this kernel increases in an exponential fashion. The unexpected spike for the data set with 32 bands is expected to be due to the relationship of this implementation and a warp-size boundary condition, though this has not been explored thoroughly.

Another significant contributing factor to the time required for the CovRX kernel is the size of the EW. A regression analysis of the trend shown by varying the EW size discloses an exponential relationship as the EW size increases. It is evident that upon optimization of the GPU code, application with a smaller EW size will realize real-time processing speeds first.

Implications for Real-time Processing

A discussion about real-time processing occasions a definition of real-time. For the purposes of this discussion the specifications of the CASI 1500 HSI sensor by ITRES Research Limited are used(ITRES 2010); it is the sensor used to collect the sample data employed for this research. The CASI 1500 collects at a data rate of 20MB/second. Processing a 20MB scene once every second for this sensor is real-time. This is 2MB more than the subsectioned scene in these timing tests. Therefore, processing the 512×512×36 image once every one second is near real-time for the CASI 1500 system.

Examining the code components discussed above it is seen that CUDA initialization, data transfer from host to device, and data transfer from device to host times are controlled by the GPU and CUDA, therefore the places to focus optimization efforts are the EW Sum and EW CovRX kernels.

Table 12 breaks out relevant timing information for the 512×512×36 image of interest using a 17×17 EW and the fastest block layout tested.

Table 12

GPU Component Processing Times for 512 × 512 × 36 Input and 17 × 17 EW

Component	Processing Time (ms)
CUDA Init	379.794
Host to Device	3.778
Result to Host	1.479
EW Sum	137.583
CovRX	6028.47
Component Time Sums	6551.104

Given a one hertz constraint, removing the time required by the CUDA initialization and data transfer components both the EW Sum and CovRX kernels must complete within 614ms if the initialization time is assessed for every frame. Since a real-time scenario can eliminate the need to initialize for every frame reducing the EW Sum and CovRX processing times so that both fit within 995ms will meet the given constraint and provide near real-time processing performance.

CHAPTER V

CONCLUSION

Results given above illustrate that it is reasonable to expect a GPU-based solution of the RX algorithm will perform faster than a serial solution. A naïve one block per pixel implementation has shown significant speed improvements as compared to a naïve serial implementation.

This research has shown that the algorithm design decisions for a GPU architecture differ in several aspects to decisions required when dividing the problem for a parallel solution on a traditional HPC system. In particular the GPU memory architecture differences necessitate different solutions. Additionally, within the CUDA / NVIDIA GPU paradigm, design modifications in some areas are needed for different compute capabilities.

Algorithm parallelization with a GPU not only has advantages over a serial solution, but has shown some advantages over traditional HPC systems, especially distributed HPC solutions. Likewise, the GPU has limitations that traditional HPC solutions do not share. GPU advantages and limitations specific to HSI processing are described below.

Although the performance times for the GPU implementation are not fast enough for real-time processing, improvements to this approach is expected to speed up the timing results such that real-time performance is feasible. Opportunities for improvement with future research are given. Additionally, application of this approach to HSI processing is discussed.

GPU Advantages

The speed advantage for processing HSI data with the RX algorithm with a parallelized solution using a GPU when compared with a serial solution is shown. Additional advantages of the GPU include parallel processing capability that is affordable, available, and accessible to the average HSI analyst. Similarly, the GPU has the potential for performing the RX algorithm processing in real-time which offers advantages over other HPC solutions for size, weight, and power considerations.

The GPU also offers programming advantages relative to traditional HPC systems. Specifically, parallelization at the thread level is context switch overhead free so that adding more workers does not add the overhead incurred by current HPC solutions. Also, requiring that all blocks in a kernel work independently offers the advantage of significant scalability. Another advantage of the GPU solution for HSI processing is the spatial memory access optimization designed into the hardware as opposed to row or column major access of non-graphics based processors.

One other advantage of note is that while somewhat complex, device global memory can be accessed as a traditional shared memory system by coding mutual exclusion constructs and thereby sharing the advantage over distributed memory systems. Even without the mutex solution the GPU global memory is accessed by all of the available processors and subsequent kernels more efficiently than memory in distributed systems.

GPU Limitations

While the option to implement a complex semaphore mutex solution for allowing different blocks to communicate is an advantage, the necessity may be considered a limitation, especially in comparison to traditional shared memory HPC systems in which memory lock mechanisms and support for synchronization of all processors in the system are available. Thus on the one hand block independence is an advantage for scalability on the other hand it is a limitation to inter-block, i.e. inter-SM, communication.

Covariance matrix calculations require 64-bit resolution and thus double precision operations unless specialized integer math is implemented. The double precision accuracy for CUDA compute capabilities prior to 2.0 is not guaranteed as support for this capability is in the early stages. Additionally the GT200 integer ALU is limited to 24-bit precision and requires emulation sequences of multiple instructions for integer arithmetic, thus the ALU does not natively support 32-bit precision, not to mention 64-bit precision arithmetic.

The most significant limitation of the GPU however is the limited shared memory available to the block via the streaming multiprocessor architecture. This limitation enforces a constraint that, for the naïve one block per pixel approach, prevents support for data sets with increasing numbers of bands. The size of the covariance matrix is $B \times B \times 8$ where B is the number of bands and 8 the number of bytes for a double precision floating point value. Without a more sophisticated solution the entire covariance matrix must reside in the block's shared memory, in addition to the other vectors needed for the RX calculation. This means that for

compute capabilities before 2.0, with only 16KB of shared memory the limit for the number of bands supported is approximately 40 bands.

Although global memory is relatively significant it is not sufficient to hold a $B \times B \times 8$ covariance matrix for every pixel in the data set as well as the raw data unless the image size is uncharacteristically small or the number of bands is small. However, if the number of bands is small, then the current solution is sufficient. An approach to use the global memory space to hold the covariance matrices for all pixels in the data set while the block threads perform the RX calculations has the following constraint:

$$(2WHB) + (4WHB) + (8WHB^2) \leq VRAM \qquad \text{Eq. 19}$$

where W and H are the input data width and height, respectively, and $VRAM$ is the device global memory size. The first term accounts for the input data consisting of 2 byte shorts, the second term relates to the EW Sum “scratch” pad of floating point values and the final term represents the covariance matrices for all pixels in the input data.

Assigning one block per pixel and allowing the covariance matrices to be stored in global memory will free up the block’s shared memory and potentially accommodate larger numbers of bands, at least for small image sizes. However, this is not efficient. The latency for so many global memory accesses could require significantly more time than the current non-optimal solution. This approach does however make it possible to create a separate kernel for the RX calculation providing an opportunity to redefine the grid layout. Alternatively,

looping over this design in a serial fashion will accommodate larger images, although at the cost of a fully parallelized solution.

The limited shared memory also requires a small EW in order to capitalize on the speed of this resource, or an implementation that forfeits the speed of the shared memory in lieu of a larger EW, as in the case of the solution presented in this research.

Future Research

Although the GPU-based methodology for implementing the RX algorithm presented is significantly faster than the serial version, several insights are given that identify areas in which this methodology can be improved. The presented EW Sum kernel block layout analysis supports the NVIDIA assertion that speed performance is improved when the memory accesses are masked by the computations. Thus, balancing the memory access latency with the computation time optimizes speed performance. A careful consideration of how many threads are used in a block can improve performance; too many or too few threads impact the processing time. Too many threads, each of which accesses global memory, overwhelm the processing resources with too many memory accesses and thus the calculations do not hide the access latency. Alternatively, too few threads do not fully occupy the processing resources and thus speed performance is negatively impacted. With this in mind, the EW Sum kernel's design offers a balance between parallelizing the computation and reuse with a running EW sum approach. Implementing a similar solution with a running

covariance matrix calculation approach may offer speed improvements by better balancing the memory access latency and computations.

An opportunity to support greater numbers of bands may be to divide the $Ax = b$ solver, currently the Cholesky Decomposition, to use a partial covariance matrix. The presented solution utilizes a serial version of the Cholesky Decomposition. A parallelized solution for this solver may offer an opportunity to use a partial covariance matrix so that the shared memory limitation is mitigated and thus offer support for larger numbers of bands. Alternatively, an iterative approach such as Gauss-Seidel may allow a solution for more bands because only a single row of the covariance matrix is needed at a time while solving for x.

Another significant opportunity for speed improvement is offered by NVIDIAs Fermi GPU with compute capability 2.0. With the Fermi, NVIDIA addresses the limitations mentioned above. As mentioned the double precision accuracy and stability is improved with this architecture. Not only is the double precision accuracy improved, the Fermi has an effective 16 double precision floating point units per SM compared to the single DPFPU in the GPUs with compute capability less than 2.0. The Fermi architecture comprises 32 SMs each of which has 16 SPs for a total 512 CUDA cores. Additionally, the compute capability 2.0 associated with the Fermi doubles the numbers of threads a block is allowed from 512 to 1024 threads per block (Brookwood 2009).

Furthermore, in contrast to the single warp issue of earlier architectures, the Fermi series provides a dual warp scheduler that allows two warps with 32 threads each to be issued and executed concurrently within an SM. Adding to

these capabilities that will benefit the RX GPU algorithm's speed performance on the GPU, hardware prediction support is available at the instruction level for compute capability 2.0. This eliminates branch overhead for short conditional code segments such as those in the Cholesky Decomposition code (Glaskowsky 2009).

More importantly however is the increased memory resources. The Fermi increases the number of 32-bit registers from 16,000 to 32,000, triples the amount of shared memory bringing it up to 48KB per SM, includes an L1 cache with 16KB that takes register spills that in earlier versions went to global memory, and allocates 512KB of local memory per thread as opposed to the 64KB offered in previous versions. Beyond this, Femi introduces an L2 cache with 768KB that benefit cases where multiple SMs read the same data, as in the case for the kernels in this solution. This L2 cache also implements atomic operations in global memory without the need for semaphores. In addition, up to 6GB of DDR5 DRAM can be connected to the chip to significantly increase the capacity with faster bandwidth (NVIDIA 2009a).

The improvements in the GPU provided by the Fermi architecture will improve the speed performance of the current solution without any modifications. However, an improved solution will maximize utilization of the increased memory capacity, in particular the on-chip shared memory. These hardware improvements provide an opportunity to improve the RX algorithm implementation that could result in real-time speed performance.

HSI Application

Given an improved GPU implementation of the RX algorithm a GPU based system designed for on-board aircraft near real-time processing is possible. Indeed, a system that uses the RX, spectral matched filter, or generalized likelihood ratio test is feasible.

This study shows that it may not be feasible to store a covariance matrix associated with every pixel in global memory for it to be persistent across kernel instantiations. Therefore, the RX calculations that require the covariance matrix are implemented in the same kernel in which the covariance matrix is completed. This methodology could also be applied for the SMF algorithm which differs from the RX only in that the first term requires a spectral signature rather than a pixel from the input data. The constant memory available in global memory and the associated SMs constant cache memory could easily be used to store the a priori spectral signature. This would be available for all blocks without incurring a significant access penalty.

Similarly, the GLRT algorithm requires an a priori spectral signature. The GLRT calculations however, while much the same form as the RX and SMF, employs the $Ax = b$ solver twice; once with the input data pixel and once with the spectral signature. Thus, although a single covariance matrix will suffice, the GLRT will require a small amount more of shared memory than the RX or SMF algorithms to hold the intermediate results.

Summary

Processing HSI data to detect different types of targets is of interest for application in fields as varied as homeland security, farming, and cancer detection. The sophisticated algorithms that provide the most accurate detection results require intensive computations that are not feasible for general purpose computing systems available today and specialized solutions are cost prohibitive and inaccessible to the average HSI analyst. At less than $2,000 and in a form factor designed to integrate with a personal computer the modern GPU is both affordable and accessible relative to FPGA or HPC solutions.

In addition to the cost effectiveness and accessibility of the GPU, advancements in GPU systems that offer more processing capability as well as programmability provide an attractive alternative solution for real-time HSI target detection processing. While the GPU architecture requires a different programming paradigm the developed methodology reflects a pattern for mapping the HSI target detection algorithm problem to the NVIDIA GPU architecture.

Although the GPU offers several advantages over both serial and traditional HPC processing systems with regard to HSI data processing, the limitations of the GPU are significant. Nevertheless the research presented here indicates that real-time HSI processing is feasible. Additional effort is required to improve the given technique for application in real-time scenarios. An optimized solution designed for a Fermi GPU with CUDA compute capability 2.0 is expected to reach this goal for the RX, SMF, and GLRT algorithms.

REFERENCES

Abazovic, Fuad. 2002. Fire GL X1 to replace CPU in rendering. *The Inquirer,* August 2. http://www.theinquirer.net/inquirer/news/1023366/fire-gl-x1-to-replace-cpu-in-rendering.

Abi-Chahla, Fedy. 2008. NVIDIA's CUDA: The end of the CPU? *Tom's Hardware,* June 18. http://www.tomshardware.com/index.php?ctrl=editorial_reviews_print&p1=1954.

Alonso, Maria C., and José A. Malpica. 2009. The combination of three statistical methods for visual inspection of anomalies in hyperspectral imageries. In *2009 Seventh International Conference on Advances in Pattern Recognition*, 377–380. Washington, DC: IEEE Computer Society. doi:10.1109/ICAPR.2009.78.

AMD. 2006. AMD "Close to Metal"™ Technology Unleashes the Power of Stream Computing. *AMD Press Release*. http://www.amd.com/us/press-releases/Pages/Press_Release_114147.aspx.

Baker, Zachary K., Maya B. Gokhale, and Justin L. Tripp. 2007. Matched filter computation on FPGA, cell and GPU. In *2007 IEEE Symposium on Field-Programmable Custom Computing Machines*, 207–218. Napa: IEEE Computer Society. doi:10.1109/FCCM.2007.52.

Banerjee, Amit, Philippe Burlina, and Chris Diehl. 2009. One-class SVMs for hyperspectral anomaly detection. In *Kernel Methods for Remote Sensing*

Data Analysis, eds. G. Camps-Valls and L. Bruzzone, 169–192. West Sussex, UK: John Wiley & Sons.

Bernabe, Sergio, Antonio Plaza, Prashanth Reddy Marpu, and Jon Atli Benediktsson. 2012. A new parallel tool for classification of remotely sensed imagery. *Computers & Geosciences* 46: 208–218. doi:10.1016/j.cageo.2011.12.009.

Bolotoff, Paul. 2010. A quick analysis of the NVIDIA Fermi architecture. *Alasir*. February 16. http://alasir.com/articles/nvidia_fermi_architecture.

Borstad. Compact airborne spectrographic imager (CASI). *Borstad Associates Ltd. - Compact Airborne Spectrographic Imager (CASI)*. http://www.borstad.com/casi.html.

Brookwood, Nathan. 2009. NVIDIA solves the GPU computing puzzle. NVIDIA. http://www.nvidia.co.jp/content/PDF/fermi_white_papers/N.Brookwood_NVIDIA_Solves_the_GPU_Computing_Puzzle.pdf.

Buck, Ian. 2004. Programming GPUs for general computing. *EE Times*. December 13. http://www.eetimes.com/electronics-news/4051105/Programming-GPUs-for-general-computing.

Buck, Ian, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. 2004. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics - Proceedings of ACM SIGGRAPH 2004* 23 (3) (August): 777–786. doi:10.1145/1015706.1015800.

Canty, Morton J., and Allan A. Nielsen. 2011. Linear and kernel methods for multivariate change detection. *Computers & Geosciences* 38 (1) (June): 107–114. doi:10.1016/j.cageo.2011.05.012.

Chang, Yang-Lang, Jyh-Perng Fang, Jon Atli Benediktsson, Lena Chang, Hsuan Ren, and Kun-Shan Chen. 2009. Band selection for hyperspectral images based on parallel particle swarm optimization schemes. In *2009 IEEE International Geoscience and Remote Sensing Symposium*, 5:84–87. Cape Town: IEEE. doi:10.1109/IGARSS.2009.5417728.

Chu, Michael M. 2010. GPU computing: Past, present and future with ATI stream technology. Event presented at the IEEE Computer Society Event, March, Mountain View, CA. http://developer.amd.com/wordpress/media/2012/10/GPU%20Computing%20-%20Past%20Present%20and%20Future%20with%20ATI%20Stream%20Technology.pdf.

CVG. 2011. CUDA memory architecture GPGPU class week 4.” Swiss Federal Institute of Technology Zurich. *Computer Vision and Geometry Group*. http://www.cvg.ethz.ch/teaching/2011spring/gpgpu/cuda_memory.pdf.

Dang, Alan. 2009. Exclusive interview: NVIDIA’s Ian Buck talks GPGPU. *Tom’s Hardware*. September 3. http://www.tomshardware.com/reviews/ian-buck-nvidia,2393.html#xtor=RSS-993.

Del Rizzo, Bryan. 2012. NVIDIA launches first GeForce GPUs based on next-generation Kepler architecture. *NVIDIA News Rlease*. March 22.

http://nvidianews.nvidia.com/Releases/NVIDIA-Launches-First-GeForce-GPUs-Based-on-Next-Generation-Kepler-Architecture-79b.aspx.

Dipert, Brian. 2005. Instigating a platform tug of war: Graphics vendors hunger for CPU suppliers' turf. *EDN*. October 13. http://www.edn.com/design/consumer/4323012/Instigating-a-platform-tug-of-war-Graphics-vendors-hunger-for-CPU-suppliers-turf.

Erskine, Dave, and Matthew Kanas. 2011. AMD launches world's fastest single-GPU graphics card – the AMD Radeon™ HD 7970. *AMD*. December 22. http://www.amd.com/us/press-releases/Pages/amd-launches-worlds-fastest-2011dec22.aspx.

Fang, Jianbin, Ana Lucia Varbanescu, and Henk Sips. 2011. A comprehensive performance comparison of CUDA and OpenCL. In *2011 International Conference on Parallel Processing*, 216 –225. Taipei: IEEE Computer Society. doi:10.1109/ICPP.2011.45.

Farber, Rob. 2009. CUDA, supercomputing for the masses: Part 13. *Dr. Dobb's the World of Software Development*. June 23. http://www.drdobbs.com/parallel/cuda-supercomputing-for-the-masses-part/218100902.

Fernando, Randima, and Mark J. Kilgard. 2003. *The Cg tutorial. The definitive guide to programmable real-time graphics*. Boston, MA: Addison-Wesley.

Filho, Abel G., Alejandro C. Frery, Cristiano C. Araujo, Haglay Alice, Jorge Cerqueira, Juliana A. Loureiro, Manoel E. Lima, Maria Oliveira, and Michelle M. Horta. 2003. Hyperspectral images clustering on

reconfigurable hardware using the K-means algorithm. In *2003 16th Symposium on Integrated Circuits and Systems Design*, 99 – 104. IEEE. doi:10.1109/SBCCI.2003.1232813.

Fresse, V., D. Houzet, and C. Gravier. 2010. GPU architecture evaluation for multispectral and hyperspectral image analysis. In *2010 Conference on Design and Architectures for Signal and Image Processing*, 121–127. Edinburgh: IEEE. doi:10.1109/DASIP.2010.5706255.

Fresse, Virginie, Dominique Houzet, and Christophe Gravier. 2011. Evaluation of CPU and GPU architectures for spectral image analysis algorithms. In *Proceedings of SPIE*. Vol. 7872. San Francisco: SPIE. doi:10.1117/12.872514.

Girouard, G., A. Bannari, A. E Harti, and A. Desrochers. 2004. Validated spectral angle mapper algorithm for geological mapping: Comparative study between QuickBird and Landsat-TM. In *XXth International Society for Photgrammetry and Remote Sensing Congress*, ed. Orhan Altan, XXXV:599–604. Technical Commission IV. Istanbul: ISPRS.

Glaskowsky, Peter N. 2009. NVIDIA’s Fermi: The first complete GPU computing architecture. NVIDIA. http://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIAFermi-TheFirstCompleteGPUComputingArchitecture.pdf.

GPGPU. About GPGPU.org. *GPGPU*. http://gpgpu.org/about.

Hager, Georg, and Gerhard Wellein. 2011. *Introduction to high performance computing for scientists and engineers*, ed. Horst Simon. Computational Science 7. Boca Raton: CRC Press.

Halfhill, Tom. 2009. Looking beyond graphics. In-Stat. http://www.nvidia.com/content/PDF/fermi_white_papers/T.Halfhill_Looking_Beyond_Graphics.pdf.

Harris, Mark. 2005. Mapping computational concepts to GPUs. In *ACM SIGGRAPH 2005 Courses*, ed. John Fujii. SIGGRAPH '05. New York: ACM. doi:10.1145/1198555.1198768.

Harsanyi, Joseph C., and Chein-i Chang. 1994. Hyperspectral image classification and dimensionality reduction: An orthogonal subspace projection approach. *IEEE Transactions on Geoscience and Remote Sensing* 32 (4) (July): 779–785. doi:10.1.1.138.5825.

Hastings, David A., and William J. Emery. 1992. The advanced very high resolution radiometer (AVHRR): A brief reference guide. *Photogrammetric Engineering and Remote Sensing* 58 (8) (August): 1183–1188. American Society for Photogrammetry and Remote Sensing.

Heras, D. B, F. Arguello, J. L Gomez, J. A Becerra, and R. J Duro. 2011. Towards real-time hyperspectral image processing, a GP-GPU implementation of target identification. In *2011 IEEE 6th International Conference on Intelligent Data Acquisition and Advanced Computing Systems*, 1:316–321. Prague: IEEE. doi:10.1109/IDAACS.2011.6072765.

Huertas, Andres, and R. Nevatia. Building detection assisted by HYDICE cues. *USC Computer Vision - HYDICE Integration*. http://iris.usc.edu/Projects/muri/hydice/hydice.html.

ITRES. 2010. CASI-1500. http://www.itres.com/assets/pdf/CASI-1500.pdf.

Jensen, John R. 2007. *Remote sensing of the environment: An earth resource perspective*. University of Minnesota: Pearson Prentice Hall.

Jia, Sen, Yuntao Qian, and Zhen Ji. 2008. Band selection for hyperspectral imagery using affinity propagation. In *2008 Digital Image Computing: Techniques and Applications*, 137 –141. Canberra: IEEE. doi:10.1109/DICTA.2008.42.

Joevivek, V., T. Hemalatha, and K.P. Soman. 2009. Determining an efficient supervised classification method for hyperspectral image. In *2009 International Conference on Advances in Recent Technologies in Communication and Computing*, 384 –386. Kottayam: IEEE. doi:10.1109/ARTCom.2009.174.

Kanter, David. 2008. NVIDIA's GT200: Inside a parallel processor. *Real World Technologies*. September 8. http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242.

Kelly, E.J. 1986. An adaptive detection algorithm. *IEEE Transactions on Aerospace and Electronic Systems* AES-22 (2) (March): 115 –127. doi:10.1109/TAES.1986.310745.

Kingyens, Jeffrey, and J. Gregory Steffan. 2011. The potential for a GPU-like overlay architecture for FPGAs. *International Journal of Reconfigurable Computing* 2011: 15. doi:10.1155/2011/514581.

Kirk, David. 2007. NVIDIA CUDA software and GPU parallel computing architecture. In *Proceedings of the 6th International Symposium on Memory Management*, 103–104. ISMM '07. New York: ACM. doi:10.1145/1296907.1296909. http://doi.acm.org/10.1145/1296907.1296909.

Kowaliski, Cyril. 2008. AMD refines its approach to stream computing and wages the GPU compute war on many fronts. *The Tech Report PC Hardware Explored*. June 19. http://techreport.com/articles.x/14968.

Kruse, F.A., A.B. Lefkoff, J.W. Boardman, K.B. Heidebrecht, A.T. Shapiro, P.J. Barloon, and A.F.H. Goetz. 1993. The spectral image processing system (SIPS) - Interactive visualization and analysis of imaging spectrometer data. *Remote Sensing of Environment* 44 (2-3) (May): 145–163. doi:10.1016/0034-4257(93)90013-N.

Kruse, Fred A. 1994. Imaging spectrometer data analysis – A tutorial. In *Proceeding of International Symposium on Spectral Sensing Research*, I:44–54. San Diego.

Kwatra, Vivek, and Mei Han. 2010. Fast covariance computation and dimensionality reduction for sub-window features in images. In *Proceedings of the 11th European Conference on Computer Vision: Part*

II, eds. Kostas Daniilidis, Petros Maragos, and Nikos Paragios, 156–169. Berlin: Springer-Verlag.

Lal Shimpi, Anand, and Derek Wilson. 2008. AnandTech - NVIDIA's 1.4 billion transistor GPU: GT200 arrives as the GeForce GTX 280 & 260. *AnandTech*. June 16. http://www.anandtech.com/show/2549.

Lillesand, Thomas M., Ralph W. Kiefer, and Jonathan W. Chipman. 2004. *Remote sensing and image interpretation,* eds. Ryan Flahive, Denise Powell, and Norine M. Pigliucci. 5th ed. USA: Wiley.

LoPiccolo, Phil. 2003. Prime prospects. *Computer Graphics World*, March. http://www.cgw.com/Publications/CGW/2003/Volume-26-Issue-3-March-2003-/Prime-Prospects.aspx.

Luo, Wenfei. 2011. Parallel implementation of N-FINDR algorithm for hyperspectral imagery on hybrid multiple-core CPU and GPU parallel platform. In *Proceedings of SPIE*. Vol. 8006. Guilin, China: SPIE. doi:10.1117/12.901593.

Macedonia, Michael. 2003. The GPU enters computing's mainstream. *Computer* 36 (10) (October): 106–108. doi:10.1109/MC.2003.1236476.

Manolakis, D., R. Lockwood, T. Cooley, and J. Jacobson. 2007. Robust matched filters for target detection in hyperspectral imaging data. In *2007 IEEE International Conference on Acoustics, Speech and Signal Processing*, 1:529–532. Honolulu: IEEE. doi:10.1109/ICASSP.2007.366733.

———. 2009. Hyperspectral detection algorithms: Use covariances or subspaces? In *Imaging Spectrometry XIV.* Vol. 7457. San Diego: SPIE. doi:10.1117/12.828397.

Martellaro, John. 2008. Khronos compute working group evaluating Apple's OpenCL. *The Mac Observer*. June 16. http://www.macobserver.com/tmo/article/Khronos_Compute_Working_Group_Evaluating_Apples_OpenCL/.

Matteoli, Stefania, Marco Diani, and Giovanni Corsini. 2010. Improved estimation of local background covariance matrix for anomaly detection in hyperspectral images. *Optical Engineering* 49 (4) (April 1): 046201. doi:10.1117/1.3386069.

NASA. 2013a. About MODIS. *MODIS Website*. Accessed March 8. http://modis.gsfc.nasa.gov/about/.

NASA, JPL. 2013b. AVIRIS - airborne visible / infrared imaging spectrometer. Accessed March 8. http://aviris.jpl.nasa.gov/.

Nasrabadi, Nasser M. 2007. Kernel-based spectral matched signal detectors for hyperspectral target detection. In *Pattern Recognition and Machine Intelligence*, eds. Ashish Ghosh, Rajat K. De, and Sankar K. Pal, 67–73. Kolkata: Springer.

———. 2009. Target detection with kernels. In *Kernel Methods for Remote Sensing Data Analysis*, eds. Gustavo Camps-Valls and Lorenzo Bruzzone, 147–168. West Sussex, UK: John Wiley & Sons.

Novasol. 2007a. miniARCHER | NovaSol. *Novasol Products & Services*. http://www.nova-sol.com/products-and-services/miniARCHER.

———. 2007b. CASE | NovaSol. *Novasol Products & Services*. http://www.nova-sol.com/products-and-services/processors/case.

NVIDIA. 2008. NVIDIA GeForce® GTX 200 GPU architectural overview second-generation unified GPU architecture for visual computing. TB-04044-001_v01. NVIDIA. http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf.

———. 2009a. NVIDIA's next generation CUDA compute architecture: Fermi V1.1. NVIDIA.

———. 2009b. NVIDIA advanced CUDA webinar memory optimizations. *NVIDIA Developer Training*. http://developer.download.nvidia.com/CUDA/training/NVIDIA_GPU_Computing_Webinars_CUDA_Memory_Optimization.pdf.

———. 2010. NVIDIA CUDA programming guide version 3.0. NVIDIA. http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf.

Office of Satellite Operations. Polar operational environmental satellite. *NOAA Satellite and Information Service*. http://www.oso.noaa.gov/poes/.

Owens, John D. 2004. GPUs tapped for general computing. *EE Times*. December 13. http://www.eetimes.com/electronics-news/4051104/GPUs-tapped-for-general-computing.

Paz, A., and A. Plaza. 2010. Cluster versus GPU implementation of an orthogonal target detection algorithm for remotely sensed hyperspectral images. In *2010 IEEE International Conference on Cluster Computing*, 227–234. IEEE. doi:10.1109/CLUSTER.2010.28.

Paz, Abel, and Antonio Plaza. 2010. GPU implementation of target and anomaly detection algorithms for remotely sensed hyperspectral image analysis. In *Proceedings of SPIE*. Vol. 7810. San Diego: SPIE. doi:10.1117/12.860213.

———. 2011. A new morphological anomaly detection algorithm for hyperspectral images and its GPU implementation. In *Proceedings of SPIE*. Vol. 8157. San Diego: SPIE. doi:10.1117/12.892282.

Peddie, Jon. 2011. Report: The state of the graphics world. *Computer Graphics World*, November 9. http://www.cgw.com/Press-Center/Online-Exclusives/2011/Report-The-State-of-the-Graphics-World.aspx.

Plaza, A., J. Plaza, and S. Sanchez. 2009. Parallel implementation of endmember extraction algorithms using NVIDIA graphical processing units. In *2009 IEEE International Geoscience and Remote Sensing Symposium*, 5:208–211. IEEE. doi:10.1109/IGARSS.2009.5417696.

Press, William, Saul Teukolsky, William Vetterling, and Brian Flannery. 1992. *Numerical recipes in C: The art of scientific computing*. 2nd ed. New York: Cambridge University Press.

Pulsone, Nicholas, and Michael Zatman. 1999. A computationally-efficient two-step implementation of the GLRT detector. In *1999 IEEE International*

Conference on Acoustics, Speech, and Signal Processing, 3:1161–1164. Phoenix: IEEE. doi:10.1109/ICASSP.1999.756183.

Rees, W.G. 2003. *Physical principles of remote sensing*. 2nd ed. Cambridge, UK: Cambridge University Press.

Robila, S.A., and G. Busardo. 2011. Hyperspectral data processing in a high performance computing environment: A parallel best band selection algorithm. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, 1424 – 1431. Shanghai: IEEE. doi:10.1109/IPDPS.2011.282.

Rosario-Torres, S., and M. Velez-Reyes. 2009. Speeding up the MATLAB™ hyperspectral image analysis toolbox using GPUs and the Jacket Toolbox. In *2009 First Workshop on Hyperspectral Image and Signal Processing: Evolution in Remote Sensing*, 1–4. Grenoble: IEEE. doi:10.1109/WHISPERS.2009.5289089.

RSI. 2003. *ENVI version 4.0 user's guide*. September, 2003. Boulder, CO: Research Systems Inc.

Ruetsch, Greg, and Brent Oster. 2008. Getting started with CUDA presented at the NVISION 08 Conference, San Jose, CA. http://www.nvidia.com/content/cudazone/download/Getting_Started_w_CUDA_Training_NVISION08.pdf.

Rys. 2007. NVIDIA G80: Architecture and GPU analysis. *Beyond3D*. April 25. http://www.beyond3d.com/content/reviews/1/1.

———. 2008. NVIDIA GT200 GPU and architecture analysis. *Beyond3D*. June 16. http://www.beyond3d.com/content/reviews/51.

Sanchez, S., and A. Plaza. 2010. GPU implementation of the pixel purity index algorithm for hyperspectral image analysis. In *2010 IEEE International Conference on Cluster Computing Workshops and Posters*, 1–7. Heraklion: IEEE. doi:10.1109/CLUSTERWKSP.2010.5613110.

Sanchez, Sergio, and Antonio Plaza. 2011a. Real-time implementation of a full hyperspectral unmixing chain on graphics processing units. In *Proceedings of SPIE*. Vol. 8157. San Diego: SPIE. doi:10.1117/12.892284.

———. 2011b. A comparative analysis of GPU implementations of spectral unmixing algorithms. In *Proceedings of SPIE*. Vol. 8183. Prague: SPIE. doi:10.1117/12.897329.

Sanchez, Sergio, Gabriel Martin, Abel Paz, Antonio Plaza, and Javier Plaza. 2010. Near real-time endmember extraction from remotely sensed hyperspectral data using NVIDIA GPUs. In *Real-Time Image and Video Processing 2010*, eds. Nasser Kehtarnavaz and Matthias F. Carlsohn. Vol. 7724. Brussels: SPIE. doi:10.1117/12.854365.

Sanchez, Sergio, Gabriel Martin, Antonio Plaza, and Chein-I Chang. 2010. GPU implementation of fully constrained linear spectral unmixing for remotely sensed hyperspectral data exploitation. In *Proceedings of SPIE*. Vol. 7810. San Diego: SPIE. doi:10.1117/12.860775.

Schott, John R. 2007. *Remote sensing: The image chain approach*. 2nd ed. New York: Oxford University Press.

Setoain, J., M. Prieto, C. Tenllado, A. Plaza, and F. Tirado. 2007. Parallel morphological endmember extraction using commodity graphics hardware. *IEEE Geoscience and Remote Sensing Letters* 4 (3) (July): 441–445. doi:10.1109/LGRS.2007.897398.

Setoain, J., C. Tenllado, M. Prieto, D. Valencia, A. Plaza, and J. Plaza. 2006. Parallel hyperspectral image processing on commodity graphics hardware. In *2006 International Conference on Parallel Processing Workshops*, 465–472. Columbus, OH: IEEE Computer Society. doi:10.1109/ICPPW.2006.60.

Sherwin, Lisa. 2007. PCI-SIG delivers PCI Express 2.0 specification. *PCI-SIG - PCI Express Press Releases*. January 15. http://www.pcisig.com/specifications/pciexpress/press.

Shippert, Peg. 2003. Introduction to hyperspectral image analysis. *Online Journal of Space Communication* (3). Research and Applications: 13.

Si, Xiaoshu, and Hong Zheng. 2010. High performance remote sensing image processing using CUDA. In *2010 Third International Symposium on Electronic Commerce and Security*, 121–125. Guangzhou: IEEE. doi:10.1109/ISECS.2010.35.

Sutter, Herb. 2005. A fundamental turn toward concurrency in software. Your free lunch will soon be over. What can you do about it? *Dr. Dobb's Journal* 30 (3) (March): 16–22.

Tarabalka, Yuliya, Trym V. Haavardsholm, Ingebjorg Kasen, and Torbjorn Skauli. 2008. Parallel processing for normal mixture models of hyperspectral data using a graphics processor. In *2008 IEEE International Geoscience and Remote Sensing Symposium*, 2:II–990–II–993. Boston: IEEE. doi:10.1109/IGARSS.2008.4779163.

Theiler, James, Bernard R. Foy, and Andrew M. Fraser. 2005. Characterizing non-Gaussian clutter and detecting weak gaseous plumes in hyperspectral imagery. In *Proceedings of SPIE.* 5806:182–193. Orlando: SPIE. doi:doi:10.1117/12.604075.

Thompson, Chris J., Sahngyun Hahn, and Mark Oskin. 2002. Using modern graphics architectures for general-purpose computing: A framework and analysis. In *2002 IEEE/ACM International Symposium on Microarchitecture*, 306–317. Istanbul: IEEE Computer Society. doi:10.1109/MICRO.2002.1176259.

Topping, Rusty. 2009. Graphics processing units revolutionize hyperspectral data processing. *SPIE Newsroom*: 2. doi:10.1117/2.1200910.1834.

Trigueros-Espinosa, Blas, Miguel Velez-Reyes, Nayda G Santiago-Santiago, and Samuel Rosario-Torres. 2011. Evaluation of the GPU architecture for the implementation of target detection algorithms for hyperspectral imagery. In *Proceedings of SPIE*. Vol. 8048. Orlando: SPIE. doi:doi:10.1117/12.885621.

Tulloch, Paul. 2006. Supercomputing’s next revolution. *WIRED*. November 9. http://www.wired.com/gadgets/pcs/news/2006/11/72090.

USGS. EO-1. *USGS EO-1 Website*. http://eo1.usgs.gov/.

Walrath, Josh. 2008. NVIDIA GT200: Moving away from just a GPU. *PC Perspective*. June 16. http://www.pcper.com/reviews/Graphics-Cards/NVIDIA-GT200-Moving-Away-Just-GPU.

West, Jason E., David W. Messinger, Emmett J. Lentilucci, John P. Kerekes, and John R. Schott. 2005. Matched filter stochastic background characterization for hyperspectral target detection. In *Proceedings of SPIE.* 5806:1–12. Orlando: SPIE. doi:10.1117/12.605727.

Winter, Michael E, and Edwin M Winter. 2011. Hyperspectral processing in graphical processing units. In *Proceedings of SPIE*. Vol. 8048. Orlando: SPIE. doi:doi:10.1117/12.884668.

Yang, He, and Qian Du. 2011. Fast band selection for hyperspectral imagery. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, 1048–1051. Tainan: IEEE. doi:10.1109/ICPADS.2011.157.

Yang, He, Qian Du, and G. Chen. 2011. Unsupervised hyperspectral band selection using graphics processing units. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 4 (3) (September): 660–668. doi:10.1109/JSTARS.2011.2120598.

Yuan, Jinguo, and Zheng Niu. 2007. Classification using EO-1 hyperion hyperspectral and ETM data. In *2007 Fourth International Conference on Fuzzy Systems and Knowledge Discovery*, 3:538 –542. Haikou: IEEE. doi:10.1109/FSKD.2007.218.

Zhang, Jian, and Kim Hwa Lim. 2011. Implementation of a covariance-based principal component analysis algorithm with a CUDA-enabled graphics processing unit. In *2011 IEEE International Geoscience and Remote Sensing Symposium*, 1759–1762. Vancouver: IEEE. doi:10.1109/IGARSS.2011.6049460.